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>
```

```
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64Links(ILinks<ulong> links) : base(links) { }

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

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

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

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

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

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

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

```
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>
```

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

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

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

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

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

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

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

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

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

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

using System.Collections.Generic;

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

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

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using Platform.Ranges;
```

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

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

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

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

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

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

```

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

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

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

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

        #region Paths

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

```
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         }
```

```

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Incrementers
{
    public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;

        private readonly TLink _frequencyMarker;
        private readonly TLink _unaryOne;
        private readonly IIncrementer<TLink> _unaryNumberIncrementer;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
        ↪ IIncrementer<TLink> unaryNumberIncrementer)
            : base(links)
        {
            _frequencyMarker = frequencyMarker;
            _unaryOne = unaryOne;
            _unaryNumberIncrementer = unaryNumberIncrementer;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Increment(TLink frequency)
        {
            var links = _links;
            if (_equalityComparer.Equals(frequency, default))
            {
                return links.GetOrCreate(_unaryOne, _frequencyMarker);
            }
            var incrementedSource =
            ↪ _unaryNumberIncrementer.Increment(links.GetSource(frequency));
            return links.GetOrCreate(incrementedSource, _frequencyMarker);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Incrementers
{
    public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;

        private readonly TLink _unaryOne;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
        ↪ _unaryOne = unaryOne;
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Increment(TLink unaryNumber)
        {
            var links = _links;
            if (_equalityComparer.Equals(unaryNumber, _unaryOne))
            {
                return links.GetOrCreate(_unaryOne, _unaryOne);
            }
            var source = links.GetSource(unaryNumber);
            var target = links.GetTarget(unaryNumber);
            if (_equalityComparer.Equals(source, target))
            {
                return links.GetOrCreate(unaryNumber, _unaryOne);
            }
            else
            {
                return links.GetOrCreate(source, Increment(target));
            }
        }
    }
}
```

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

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

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

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

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

        private const int Length = 3;

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
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)
58          {
59              Index = index;
60              Source = source;
61              Target = target;
62          }
63
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
            ↪  out TLink target)
66          {
67              index = other.Index;
68              source = other.Source;
69              target = other.Target;
70          }
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          private static void SetValues(IList<TLink> values, out TLink index, out TLink source,
            ↪  out TLink target)
74          {
75              switch (values.Count)
76              {
77                  case 3:
78                      index = values[0];
79                      source = values[1];
80                      target = values[2];
81                      break;
82                  case 2:
83                      index = values[0];
84                      source = values[1];
85                      target = default;
86                      break;
87                  case 1:
88                      index = values[0];
89                      source = default;
90                      target = default;
91                      break;
92                  default:
93                      index = default;
94                      source = default;
95                      target = default;
96                      break;
97              }
98          }
99
100         [MethodImpl(MethodImplOptions.AggressiveInlining)]
101         public override int GetHashCode() => (Index, Source, Target).GetHashCode();
102
103         [MethodImpl(MethodImplOptions.AggressiveInlining)]
104         public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
105                             && _equalityComparer.Equals(Source, _constants.Null)
106                             && _equalityComparer.Equals(Target, _constants.Null);
107
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         public override bool Equals(object other) => other is Link<TLink> &&
            ↪  Equals((Link<TLink>)other);
110
111         [MethodImpl(MethodImplOptions.AggressiveInlining)]
112         public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
113                                               && _equalityComparer.Equals(Source, other.Source)
114                                               && _equalityComparer.Equals(Target, other.Target);
115
116         [MethodImpl(MethodImplOptions.AggressiveInlining)]
117         public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
            ↪  {source}->{target})";
118
119         [MethodImpl(MethodImplOptions.AggressiveInlining)]
120         public static string ToString(TLink source, TLink target) => $"({source}->{target})";
121
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         public static implicit operator TLink[](Link<TLink> link) => link.ToArray();
124
125         [MethodImpl(MethodImplOptions.AggressiveInlining)]
126         public static implicit operator Link<TLink>(TLink[] linkArray) => new
            ↪  Link<TLink>(linkArray);
127
```

```
128         [MethodImpl(MethodImplOptions.AggressiveInlining)]
129         public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
            ↪ ToString(Source, Target) : ToString(Index, Source, Target);
130
131         #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 }
```

1.25 ./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 }
```

1.26 ./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 }
```

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

```
using System.Runtime.CompilerServices;

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

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

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

1.28 ./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);
    }
}
```

1.29 ./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
    }
}
```

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

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

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

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

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

        public TLink AllocatedLinks;
        public TLink ReservedLinks;
        public TLink FreeLinks;
        public TLink FirstFreeLink;
```

```
        public TLink RootAsSource;
        public TLink RootAsTarget;
        public TLink LastFreeLink;
        public TLink Reserved8;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

```
            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);
        }
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

```
10          public ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
            ↪  constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
            ↪  base(constants, linksDataParts, linksIndexParts, header) { }
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          protected override ref TLink GetLeftReference(TLink node) => ref
            ↪  GetLinkIndexPartReference(node).LeftAsSource;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected override ref TLink GetRightReference(TLink node) => ref
            ↪  GetLinkIndexPartReference(node).RightAsSource;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          protected override TLink GetLeft(TLink node) =>
            ↪  GetLinkIndexPartReference(node).LeftAsSource;
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          protected override TLink GetRight(TLink node) =>
            ↪  GetLinkIndexPartReference(node).RightAsSource;
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override void SetLeft(TLink node, TLink left) =>
            ↪  GetLinkIndexPartReference(node).LeftAsSource = left;
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override void SetRight(TLink node, TLink right) =>
            ↪  GetLinkIndexPartReference(node).RightAsSource = right;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override TLink GetSize(TLink node) =>
            ↪  GetLinkIndexPartReference(node).SizeAsSource;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override void SetSize(TLink node, TLink size) =>
            ↪  GetLinkIndexPartReference(node).SizeAsSource = size;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override TLink GetBasePartValue(TLink link) =>
            ↪  GetLinkDataPartReference(link).Source;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
            ↪  (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
            ↪  (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override void ClearNode(TLink node)
50          {
51              ref var link = ref GetLinkIndexPartReference(node);
52              link.LeftAsSource = Zero;
53              link.RightAsSource = Zero;
54              link.SizeAsSource = Zero;
55          }
56      }
57  }
```

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

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

```
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          protected override ref TLink GetLeftReference(TLink node) => ref
            ↪ GetLinkIndexPartReference(node).LeftAsSource;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected override ref TLink GetRightReference(TLink node) => ref
            ↪ GetLinkIndexPartReference(node).RightAsSource;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          protected override TLink GetLeft(TLink node) =>
            ↪ GetLinkIndexPartReference(node).LeftAsSource;
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          protected override TLink GetRight(TLink node) =>
            ↪ GetLinkIndexPartReference(node).RightAsSource;
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override void SetLeft(TLink node, TLink left) =>
            ↪ GetLinkIndexPartReference(node).LeftAsSource = left;
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override void SetRight(TLink node, TLink right) =>
            ↪ GetLinkIndexPartReference(node).RightAsSource = right;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override TLink GetSize(TLink node) =>
            ↪ GetLinkIndexPartReference(node).SizeAsSource;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override void SetSize(TLink node, TLink size) =>
            ↪ GetLinkIndexPartReference(node).SizeAsSource = size;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override TLink GetBasePartValue(TLink link) =>
            ↪ GetLinkDataPartReference(link).Source;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
            ↪ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
            ↪ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override void ClearNode(TLink node)
50          {
51              ref var link = ref GetLinkIndexPartReference(node);
52              link.LeftAsSource = Zero;
53              link.RightAsSource = Zero;
54              link.SizeAsSource = Zero;
55          }
56      }
57  }
```

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

```
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 ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink> :
        ↪ ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
8       {
9           [MethodImpl(MethodImplOptions.AggressiveInlining)]
10          public ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(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).LeftAsTarget;

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

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

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

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

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

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

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

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

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

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

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

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

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

```
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;
        }
    }
}
```

1.37 ./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;
```

```
21          protected readonly byte* Header;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected InternalLinksRecursionlessSizeBalancedTreeMethodsBase(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(TLink link);
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected abstract TLink GetBasePartValue(TLink link);
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected abstract TLink GetKeyPartValue(TLink link);
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
            ↪  AsRef<RawLinkDataPart<TLink>>(LinksDataParts + (RawLinkDataPart<TLink>.SizeInBytes *
            ↪  _addressToInt64Converter.Convert(link)));
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
            ↪  ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
            ↪  (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
            ↪  LessThan(GetKeyPartValue(first), GetKeyPartValue(second));
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
            ↪  GreaterThan(GetKeyPartValue(first), GetKeyPartValue(second));
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          public TLink this[TLink link, TLink index]
62          {
63              [MethodImpl(MethodImplOptions.AggressiveInlining)]
64              get
65              {
66                  var root = GetTreeRoot(link);
67                  if (GreaterOrEqualThan(index, GetSize(root)))
68                  {
69                      return Zero;
70                  }
71                  while (!EqualToZero(root))
72                  {
73                      var left = GetLeftOrDefault(root);
74                      var leftSize = GetSizeOrZero(left);
75                      if (LessThan(index, leftSize))
76                      {
77                          root = left;
78                          continue;
79                      }
80                      if (AreEqual(index, leftSize))
81                      {
82                          return root;
83                      }
84                      root = GetRightOrDefault(root);
85                      index = Subtract(index, Increment(leftSize));
86                  }
87                  return Zero; // TODO: Impossible situation exception (only if tree structure
                    ↪  broken)
88              }
89          }
90
91          /// <summary>
```

```
92          /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
            ↪  (концом).
93          /// </summary>
94          /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
95          /// <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 }
```

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

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

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

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

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

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

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

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

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

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

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

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

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

        public TLink this[TLink link, TLink index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                var root = GetTreeRoot(link);
                if (GreaterOrEqualThan(index, GetSize(root)))
                {
                    return Zero;
                }
```

```
            while (!EqualToZero(root))
            {
                var left = GetLeftOrDefault(root);
                var leftSize = GetSizeOrZero(left);
                if (LessThan(index, leftSize))
                {
                    root = left;
                    continue;
                }
                if (AreEqual(index, leftSize))
                {
                    return root;
                }
                root = GetRightOrDefault(root);
                index = Subtract(index, Increment(leftSize));
            }
            return Zero; // TODO: Impossible situation exception (only if tree structure broken)
        }
    }

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

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

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

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

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

```
            }
            if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
            {
                return @break;
            }
            return @continue;
        }

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

1.39 ./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);
            }
        }
```

```
51
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
53          protected override TLink GetPrevious(TLink element) =>
            ↪ GetLinkIndexPartReference(element).LeftAsSource;
54
55          [MethodImpl(MethodImplOptions.AggressiveInlining)]
56          protected override TLink GetNext(TLink element) =>
            ↪ GetLinkIndexPartReference(element).RightAsSource;
57
58          [MethodImpl(MethodImplOptions.AggressiveInlining)]
59          protected override TLink GetSize(TLink head) =>
            ↪ GetLinkIndexPartReference(head).SizeAsSource;
60
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          protected override void SetFirst(TLink head, TLink element) =>
            ↪ GetLinkIndexPartReference(head).RootAsSource = element;
63
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          protected override void SetLast(TLink head, TLink element)
66          {
67              //var first = GetLinkIndexPartReference(head).RootAsSource;
68              //if (EqualToZero(first))
69              //{
70              //    SetFirst(head, element);
71              //}
72              //else
73              //{
74              //    SetPrevious(first, element);
75              //}
76          }
77
78          [MethodImpl(MethodImplOptions.AggressiveInlining)]
79          protected override void SetPrevious(TLink element, TLink previous) =>
            ↪ GetLinkIndexPartReference(element).LeftAsSource = previous;
80
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          protected override void SetNext(TLink element, TLink next) =>
            ↪ GetLinkIndexPartReference(element).RightAsSource = next;
83
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected override void SetSize(TLink head, TLink size) =>
            ↪ GetLinkIndexPartReference(head).SizeAsSource = size;
86
87          [MethodImpl(MethodImplOptions.AggressiveInlining)]
88          public TLink CountUsages(TLink head) => GetSize(head);
89
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
92          {
93              ref var link = ref GetLinkDataPartReference(linkIndex);
94              return new Link<TLink>(linkIndex, link.Source, link.Target);
95          }
96
97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
98          public TLink EachUsage(TLink source, Func<IList<TLink>, TLink> handler)
99          {
100             var @continue = Continue;
101             var @break = Break;
102             var current = GetFirst(source);
103             var first = current;
104             while (!EqualToZero(current))
105             {
106                 if (AreEqual(handler(GetLinkValues(current)), @break))
107                 {
108                     return @break;
109                 }
110                 current = GetNext(current);
111                 if (AreEqual(current, first))
112                 {
113                     return @continue;
114                 }
115             }
116             return @continue;
117         }
118     }
119 }
```

1.40 ./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);
    }
}
```

1.41 ./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);
    }
}
```

1.42 ./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);
    }
}
```

1.43 ./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) { }

```

```
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          protected override ref TLink GetLeftReference(TLink node) => ref
            ↪ GetLinkIndexPartReference(node).LeftAsTarget;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected override ref TLink GetRightReference(TLink node) => ref
            ↪ GetLinkIndexPartReference(node).RightAsTarget;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          protected override TLink GetLeft(TLink node) =>
            ↪ GetLinkIndexPartReference(node).LeftAsTarget;
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          protected override TLink GetRight(TLink node) =>
            ↪ GetLinkIndexPartReference(node).RightAsTarget;
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override void SetLeft(TLink node, TLink left) =>
            ↪ GetLinkIndexPartReference(node).LeftAsTarget = left;
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override void SetRight(TLink node, TLink right) =>
            ↪ GetLinkIndexPartReference(node).RightAsTarget = right;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override TLink GetSize(TLink node) =>
            ↪ GetLinkIndexPartReference(node).SizeAsTarget;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override void SetSize(TLink node, TLink size) =>
            ↪ GetLinkIndexPartReference(node).SizeAsTarget = size;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override TLink GetTreeRoot(TLink link) =>
            ↪ GetLinkIndexPartReference(link).RootAsTarget;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override TLink GetBasePartValue(TLink link) =>
            ↪ GetLinkDataPartReference(link).Target;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override TLink GetKeyPartValue(TLink link) =>
            ↪ GetLinkDataPartReference(link).Source;
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override void ClearNode(TLink node)
47          {
48              ref var link = ref GetLinkIndexPartReference(node);
49              link.LeftAsTarget = Zero;
50              link.RightAsTarget = Zero;
51              link.SizeAsTarget = Zero;
52          }
53
54          public override TLink Search(TLink source, TLink target) =>
            ↪ SearchCore(GetTreeRoot(target), source);
55      }
56  }
```

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

```
1   using System;
2   using System.Runtime.CompilerServices;
3   using Platform.Singletons;
4   using Platform.Memory;
5   using static System.Runtime.CompilerServices.Unsafe;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Memory.Split.Generic
10  {
11      public unsafe class SplitMemoryLinks<TLink> : SplitMemoryLinksBase<TLink>
12      {
13          private readonly Func<ILinksTreeMethods<TLink>> _createInternalSourceTreeMethods;
14          private readonly Func<ILinksTreeMethods<TLink>> _createExternalSourceTreeMethods;
15          private readonly Func<ILinksTreeMethods<TLink>> _createInternalTargetTreeMethods;
16          private readonly Func<ILinksTreeMethods<TLink>> _createExternalTargetTreeMethods;
17          private byte* _header;
18          private byte* _linksDataParts;
19          private byte* _linksIndexParts;
20
```

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

```
            ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
            InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
            ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
            UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_linksDataParts, _header);
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        protected InternalLinksSourcesLinkedListMethods<TLink> InternalSourcesListMethods;
        protected ILinksTreeMethods<TLink> InternalSourcesTreeMethods;
```

```
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
```

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Count(IList<TLink> restrictions)
        {
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
            if (restrictions.Count == 0)
            {
                return Total;
            }
            var constants = Constants;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Total;
                }
                return Exists(index) ? GetOne() : GetZero();
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Total; // Any - как отсутствие ограничения
                    }
                    var externalReferencesRange = constants.ExternalReferencesRange;
                    if (externalReferencesRange.HasValue &&
                    ↪  externalReferencesRange.Value.Contains(value))
                    {
                        return Add(ExternalSourcesTreeMethods.CountUsages(value),
                        ↪  ExternalTargetsTreeMethods.CountUsages(value));
                    }
                    else
                    {
                        if (_useLinkedList)
                        {
                            return Add(InternalSourcesListMethods.CountUsages(value),
                            ↪  InternalTargetsTreeMethods.CountUsages(value));
                        }
                        else
```

```
                    {
                        return Add(InternalSourcesTreeMethods.CountUsages(value),
                        ↪  InternalTargetsTreeMethods.CountUsages(value));
                    }
                }
            }
            else
            {
                if (!Exists(index))
                {
                    return GetZero();
                }
                if (AreEqual(value, any))
                {
                    return GetOne();
                }
                ref var storedLinkValue = ref GetLinkDataPartReference(index);
                if (AreEqual(storedLinkValue.Source, value) ||
                ↪  AreEqual(storedLinkValue.Target, value))
                {
                    return GetOne();
                }
                return GetZero();
            }
        }
        if (restrictions.Count == 3)
        {
            var externalReferencesRange = constants.ExternalReferencesRange;
            var source = restrictions[constants.SourcePart];
            var target = restrictions[constants.TargetPart];
            if (AreEqual(index, any))
            {
                if (AreEqual(source, any) && AreEqual(target, any))
                {
                    return Total;
                }
                else if (AreEqual(source, any))
                {
                    if (externalReferencesRange.HasValue &&
                    ↪  externalReferencesRange.Value.Contains(target))
                    {
                        return ExternalTargetsTreeMethods.CountUsages(target);
                    }
                    else
                    {
                        return InternalTargetsTreeMethods.CountUsages(target);
                    }
                }
                else if (AreEqual(target, any))
                {
                    if (externalReferencesRange.HasValue &&
                    ↪  externalReferencesRange.Value.Contains(source))
                    {
                        return ExternalSourcesTreeMethods.CountUsages(source);
                    }
                    else
                    {
                        if (_useLinkedList)
                        {
                            return InternalSourcesListMethods.CountUsages(source);
                        }
                        else
                        {
                            return InternalSourcesTreeMethods.CountUsages(source);
                        }
                    }
                }
                else //if(source != Any && target != Any)
                {
                    // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
                    TLink link;
                    if (externalReferencesRange.HasValue)
                    {
                        if (externalReferencesRange.Value.Contains(source) &&
                        ↪  externalReferencesRange.Value.Contains(target))
                        {
                            link = ExternalSourcesTreeMethods.Search(source, target);
                        }
```

```
                        else if (externalReferencesRange.Value.Contains(source))
                        {
                            link = InternalTargetsTreeMethods.Search(source, target);
                        }
                        else if (externalReferencesRange.Value.Contains(target))
                        {
                            if (_useLinkedList)
                            {
                                link = ExternalSourcesTreeMethods.Search(source, target);
                            }
                            else
                            {
                                link = InternalSourcesTreeMethods.Search(source, target);
                            }
                        }
                        else
                        {
                            if (_useLinkedList ||
                                ↪ GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                                ↪ InternalTargetsTreeMethods.CountUsages(target)))
                            {
                                link = InternalTargetsTreeMethods.Search(source, target);
                            }
                            else
                            {
                                link = InternalSourcesTreeMethods.Search(source, target);
                            }
                        }
                    }
                    else
                    {
                        if (_useLinkedList ||
                            ↪ GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                            ↪ InternalTargetsTreeMethods.CountUsages(target)))
                        {
                            link = InternalTargetsTreeMethods.Search(source, target);
                        }
                        else
                        {
                            link = InternalSourcesTreeMethods.Search(source, target);
                        }
                    }
                    return AreEqual(link, constants.Null) ? GetZero() : GetOne();
                }
            }
            else
            {
                if (!Exists(index))
                {
                    return GetZero();
                }
                if (AreEqual(source, any) && AreEqual(target, any))
                {
                    return GetOne();
                }
                ref var storedLinkValue = ref GetLinkDataPartReference(index);
                if (!AreEqual(source, any) && !AreEqual(target, any))
                {
                    if (AreEqual(storedLinkValue.Source, source) &&
                        ↪ AreEqual(storedLinkValue.Target, target))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
                var value = default(TLink);
                if (AreEqual(source, any))
                {
                    value = target;
                }
                if (AreEqual(target, any))
                {
                    value = source;
                }
                if (AreEqual(storedLinkValue.Source, value) ||
                    ↪ AreEqual(storedLinkValue.Target, value))
                {
                    return GetOne();
```

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

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

```
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
```

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

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

```
549                 if (externalReferencesRange.HasValue &&
                    ↪ externalReferencesRange.Value.Contains(source))
550                 {
551                     ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
552                 }
553                 else
554                 {
555                     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 }
```

1.46 ./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  }
```

1.47 ./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  }
```

1.48 ./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>>
```

```
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

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

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

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

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

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

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

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

1.49 ./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;
```

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

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

```
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
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

1.52 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/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;
57              link.SizeAsSource = Zero;
58          }
59      }
60  }
```

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

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

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

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

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

```
            link.SizeAsTarget = Zero;
        }
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

1.56 ./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)]
```

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

1.57 ./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  }
```

1.58 ./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)]
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.59 ./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);
    }
}
```

1.60 ./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);
    }
}
```

1.61 ./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);
    }
}
```

1.62 ./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);
```

```
37                  _createExternalSourceTreeMethods = () => new
                    ↪  UInt32ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
38                  _createInternalTargetTreeMethods = () => new
                    ↪  UInt32InternalLinksTargetsSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
39                  _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)]
105         protected override TLink GetZero() => 0U;
106
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override TLink GetOne() => 1U;
109
110         [MethodImpl(MethodImplOptions.AggressiveInlining)]
111         protected override long ConvertToInt64(TLink value) => value;
112
113         [MethodImpl(MethodImplOptions.AggressiveInlining)]
114         protected override TLink ConvertToAddress(long value) => (TLink)value;
115
116         [MethodImpl(MethodImplOptions.AggressiveInlining)]
117         protected override TLink Add(TLink first, TLink second) => first + second;
118
119         [MethodImpl(MethodImplOptions.AggressiveInlining)]
120         protected override TLink Subtract(TLink first, TLink second) => first - second;
121
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         protected override TLink Increment(TLink link) => ++link;
124
125         [MethodImpl(MethodImplOptions.AggressiveInlining)]
126         protected override TLink Decrement(TLink link) => --link;
127     }
128 }
```

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

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

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

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

```
13          protected new readonly LinksHeader<TLink>* Header;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected
            ↪  UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(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)]
55          protected override bool LessThan(ulong first, ulong second) => first < second;
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          protected override ulong Increment(ulong value) => ++value;
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected override ulong Decrement(ulong value) => --value;
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected override ulong Add(ulong first, ulong second) => first + second;
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected override ulong Subtract(ulong first, ulong second) => first - second;
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override ref 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);
```

```
        }

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

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

```
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)]
```

```
61          protected override ulong Decrement(ulong value) => --value;
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected override ulong Add(ulong first, ulong second) => first + second;
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected override ulong Subtract(ulong first, ulong second) => first - second;
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override ref 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  }
```

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

```
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 UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
        ↪  UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public
            ↪  UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(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
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪ UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
        ↪ UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
        ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
        ↪ linksIndexParts, header) { }

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

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

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

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

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

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

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

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

```

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

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

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

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

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

1.69 ./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)]
```

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

1.70 ./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)]
```

```
        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);
    }
}
```

1.71 ./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);
    }
}
```

1.72 ./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];
    }
}
```

1.73 ./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);
    }
}
```

1.74 ./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);
    }
}
```

1.75 ./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);
    }
}
```

1.76 ./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;

```

```
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override ref ulong GetRightReference(ulong 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(TLink node) => LinksIndexParts[node].RootAsTarget;
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Source;
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          protected override void ClearNode(TLink node)
48          {
49              ref var link = ref LinksIndexParts[node];
50              link.LeftAsTarget = Zero;
51              link.RightAsTarget = Zero;
52              link.SizeAsTarget = Zero;
53          }
54
55          public override TLink Search(TLink source, TLink target) =>
            ↪  SearchCore(GetTreeRoot(target), source);
56      }
57  }
```

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

```
1   using System;
2   using System.Runtime.CompilerServices;
3   using Platform.Singletons;
4   using Platform.Memory;
5   using Platform.Data.Doublets.Memory.Split.Generic;
6   using TLink = System.UInt64;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets.Memory.Split.Specific
11  {
12      public unsafe class UInt64SplitMemoryLinks : SplitMemoryLinksBase<TLink>
13      {
14          private readonly Func<ILinksTreeMethods<TLink>> _createInternalSourceTreeMethods;
15          private readonly Func<ILinksTreeMethods<TLink>> _createExternalSourceTreeMethods;
16          private readonly Func<ILinksTreeMethods<TLink>> _createInternalTargetTreeMethods;
17          private readonly Func<ILinksTreeMethods<TLink>> _createExternalTargetTreeMethods;
18          private LinksHeader<ulong>* _header;
19          private RawLinkDataPart<ulong>* _linksDataParts;
20          private RawLinkIndexPart<ulong>* _linksIndexParts;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
            ↪  memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
            ↪  IndexTreeType.Default, useLinkedList: true) { }
27
```

```
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
            ↪  this(dataMemory, indexMemory, memoryReservationStep, constants,
            ↪  IndexTreeType.Default, useLinkedList: true) { }
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep, LinksConstants<TLink> constants,
            ↪  IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
            ↪  memoryReservationStep, constants, useLinkedList)
33          {
34              if (indexTreeType == IndexTreeType.SizeBalancedTree)
35              {
36                  _createInternalSourceTreeMethods = () => new
                    ↪  UInt64InternalLinksSourcesSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
37                  _createExternalSourceTreeMethods = () => new
                    ↪  UInt64ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
38                  _createInternalTargetTreeMethods = () => new
                    ↪  UInt64InternalLinksTargetsSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
39                  _createExternalTargetTreeMethods = () => new
                    ↪  UInt64ExternalLinksTargetsSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
40              }
41              else
42              {
43                  _createInternalSourceTreeMethods = () => new
                    ↪  UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
44                  _createExternalSourceTreeMethods = () => new
                    ↪  UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
45                  _createInternalTargetTreeMethods = () => new
                    ↪  UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
46                  _createExternalTargetTreeMethods = () => new
                    ↪  UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
47              }
48              Init(dataMemory, indexMemory);
49          }
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
```

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetOne() => 1UL;

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
            ref var firstLink = ref GetLinkReference(first);
            ref var secondLink = ref GetLinkReference(second);
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
            ↪  secondLink.Source, secondLink.Target);
        }

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual sbyte GetBalanceValue(TLink storedValue)
        {
            unchecked
```

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

```
208                 else // node.Key == root.Key
209                 {
210                     return root;
211                 }
212             }
213             return Zero;
214         }
215 
216         // TODO: Return indices range instead of references count
217         [MethodImpl(MethodImplOptions.AggressiveInlining)]
218         public TLink CountUsages(TLink link)
219         {
220             var root = GetTreeRoot();
221             var total = GetSize(root);
222             var totalRightIgnore = Zero;
223             while (!EqualToZero(root))
224             {
225                 var @base = GetBasePartValue(root);
226                 if (LessOrEqualThan(@base, link))
227                 {
228                     root = GetRightOrDefault(root);
229                 }
230                 else
231                 {
232                     totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
233                     root = GetLeftOrDefault(root);
234                 }
235             }
236             root = GetTreeRoot();
237             var totalLeftIgnore = Zero;
238             while (!EqualToZero(root))
239             {
240                 var @base = GetBasePartValue(root);
241                 if (GreaterOrEqualThan(@base, link))
242                 {
243                     root = GetLeftOrDefault(root);
244                 }
245                 else
246                 {
247                     totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
248 
249                     root = GetRightOrDefault(root);
250                 }
251             }
252             return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
253         }
254 
255         [MethodImpl(MethodImplOptions.AggressiveInlining)]
256         public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
257         {
258             var root = GetTreeRoot();
259             if (EqualToZero(root))
260             {
261                 return Continue;
262             }
263             TLink first = Zero, current = root;
264             while (!EqualToZero(current))
265             {
266                 var @base = GetBasePartValue(current);
267                 if (GreaterOrEqualThan(@base, link))
268                 {
269                     if (AreEqual(@base, link))
270                     {
271                         first = current;
272                     }
273                     current = GetLeftOrDefault(current);
274                 }
275                 else
276                 {
277                     current = GetRightOrDefault(current);
278                 }
279             }
280             if (!EqualToZero(first))
281             {
282                 current = first;
283                 while (true)
284                 {
285                     if (AreEqual(handler(GetLinkValues(current)), Break))
286                     {
```

```
                        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);
        }
    }
}
```

1.80 ./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)]
```

```
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
```

```
                {
                    root = GetLeftOrDefault(root);
                }
                else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                ↪  node.Key > root.Key
                {
                    root = GetRightOrDefault(root);
                }
                else // node.Key == root.Key
                {
                    return root;
                }
            }
            return Zero;
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

```
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;
```

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

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

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

```
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)]
```

```
28          protected override void SetRight(TLink node, TLink right) =>
            ↪ GetLinkReference(node).RightAsSource = right;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override TLink GetSize(TLink node) =>
            ↪ GetSizeValue(GetLinkReference(node).SizeAsSource);
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          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  }
```

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

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Memory.United.Generic
6   {
7       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)]
22          protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override void SetLeft(TLink node, TLink left) =>
            ↪  GetLinkReference(node).LeftAsSource = left;
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override void SetRight(TLink node, TLink right) =>
            ↪  GetLinkReference(node).RightAsSource = right;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsSource;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override void SetSize(TLink node, TLink size) =>
            ↪  GetLinkReference(node).SizeAsSource = size;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
            ↪  (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
            ↪  (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override void ClearNode(TLink node)
50          {
51              ref var link = ref GetLinkReference(node);
52              link.LeftAsSource = Zero;
53              link.RightAsSource = Zero;
54              link.SizeAsSource = Zero;
55          }
56      }
57  }
```

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
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;
        }
    }
}
```

1.87 ./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;
```

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

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

```
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  }
```

1.89 ./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>
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 LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
31
32          public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
33
34          public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
35
36          protected readonly IResizableDirectMemory _memory;
37          protected readonly long _memoryReservationStep;
38
39          protected ILinksTreeMethods<TLink> TargetsTreeMethods;
40          protected ILinksTreeMethods<TLink> SourcesTreeMethods;
41          // 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
42          protected ILinksListMethods<TLink> UnusedLinksListMethods;
43
44          /// <summary>
45          /// Возвращает общее число связей находящихся в хранилище.
46          /// </summary>
47          protected virtual TLink Total
48          {
49              [MethodImpl(MethodImplOptions.AggressiveInlining)]
50              get
51              {
52                  ref var header = ref GetHeaderReference();
53                  return Subtract(header.AllocatedLinks, header.FreeLinks);
54              }
55          }
56
57          public virtual LinksConstants<TLink> Constants
58          {
59              [MethodImpl(MethodImplOptions.AggressiveInlining)]
60              get;
61          }
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
            ↪  memoryReservationStep, LinksConstants<TLink> constants)
65          {
66              _memory = memory;
67              _memoryReservationStep = memoryReservationStep;
68              Constants = constants;
69          }
70
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
            ↪  memoryReservationStep) : this(memory, memoryReservationStep,
            ↪  Default<LinksConstants<TLink>>.Instance) { }
73
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
76          {
77              if (memory.ReservedCapacity < memoryReservationStep)
78              {
79                  memory.ReservedCapacity = memoryReservationStep;
80              }
81              SetPointers(memory);
82              ref var header = ref GetHeaderReference();
83              // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
84              memory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes) +
                ↪  LinkHeaderSizeInBytes;
85              // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
86              header.ReservedLinks = ConvertToAddress((memory.ReservedCapacity -
                ↪  LinkHeaderSizeInBytes) / LinkSizeInBytes);
87          }
88
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          public virtual TLink Count(IList<TLink> restrictions)
91          {
```

```
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
            if (restrictions.Count == 0)
            {
                return Total;
            }
            var constants = Constants;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Total;
                }
                return Exists(index) ? GetOne() : GetZero();
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Total; // Any - как отсутствие ограничения
                    }
                    return Add(SourcesTreeMethods.CountUsages(value),
                    ↪  TargetsTreeMethods.CountUsages(value));
                }
                else
                {
                    if (!Exists(index))
                    {
                        return GetZero();
                    }
                    if (AreEqual(value, any))
                    {
                        return GetOne();
                    }
                    ref var storedLinkValue = ref GetLinkReference(index);
                    if (AreEqual(storedLinkValue.Source, value) ||
                    ↪  AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            if (restrictions.Count == 3)
            {
                var source = restrictions[constants.SourcePart];
                var target = restrictions[constants.TargetPart];
                if (AreEqual(index, any))
                {
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return Total;
                    }
                    else if (AreEqual(source, any))
                    {
                        return TargetsTreeMethods.CountUsages(target);
                    }
                    else if (AreEqual(target, any))
                    {
                        return SourcesTreeMethods.CountUsages(source);
                    }
                    else //if(source != Any && target != Any)
                    {
                        // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
                        var link = SourcesTreeMethods.Search(source, target);
                        return AreEqual(link, constants.Null) ? GetZero() : GetOne();
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return GetZero();
                    }
```

```
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return GetOne();
                    }
                    ref var storedLinkValue = ref GetLinkReference(index);
                    if (!AreEqual(source, any) && !AreEqual(target, any))
                    {
                        if (AreEqual(storedLinkValue.Source, source) &&
                        ↪  AreEqual(storedLinkValue.Target, target))
                        {
                            return GetOne();
                        }
                        return GetZero();
                    }
                    var value = default(TLink);
                    if (AreEqual(source, any))
                    {
                        value = target;
                    }
                    if (AreEqual(target, any))
                    {
                        value = source;
                    }
                    if (AreEqual(storedLinkValue.Source, value) ||
                    ↪  AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪  поддерживаются.");
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            var constants = Constants;
            var @break = constants.Break;
            if (restrictions.Count == 0)
            {
                for (var link = GetOne(); LessOrEqualThan(link,
                ↪  GetHeaderReference().AllocatedLinks); link = Increment(link))
                {
                    if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
                    {
                        return @break;
                    }
                }
                return @break;
            }
            var @continue = constants.Continue;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Each(handler, Array.Empty<TLink>());
                }
                if (!Exists(index))
                {
                    return @continue;
                }
                return handler(GetLinkStruct(index));
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Each(handler, Array.Empty<TLink>());
                    }
                    if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
                    {
```

```
                    return @break;
                }
                return Each(handler, new Link<TLink>(index, any, value));
            }
            else
            {
                if (!Exists(index))
                {
                    return @continue;
                }
                if (AreEqual(value, any))
                {
                    return handler(GetLinkStruct(index));
                }
                ref var storedLinkValue = ref GetLinkReference(index);
                if (AreEqual(storedLinkValue.Source, value) ||
                    AreEqual(storedLinkValue.Target, value))
                {
                    return handler(GetLinkStruct(index));
                }
                return @continue;
            }
        }
        if (restrictions.Count == 3)
        {
            var source = restrictions[constants.SourcePart];
            var target = restrictions[constants.TargetPart];
            if (AreEqual(index, any))
            {
                if (AreEqual(source, any) && AreEqual(target, any))
                {
                    return Each(handler, Array.Empty<TLink>());
                }
                else if (AreEqual(source, any))
                {
                    return TargetsTreeMethods.EachUsage(target, handler);
                }
                else if (AreEqual(target, any))
                {
                    return SourcesTreeMethods.EachUsage(source, handler);
                }
                else //if(source != Any && target != Any)
                {
                    var link = SourcesTreeMethods.Search(source, target);
                    return AreEqual(link, constants.Null) ? @continue :
                    ↪  handler(GetLinkStruct(link));
                }
            }
            else
            {
                if (!Exists(index))
                {
                    return @continue;
                }
                if (AreEqual(source, any) && AreEqual(target, any))
                {
                    return handler(GetLinkStruct(index));
                }
                ref var storedLinkValue = ref GetLinkReference(index);
                if (!AreEqual(source, any) && !AreEqual(target, any))
                {
                    if (AreEqual(storedLinkValue.Source, source) &&
                        AreEqual(storedLinkValue.Target, target))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @continue;
                }
                var value = default(TLink);
                if (AreEqual(source, any))
                {
                    value = target;
                }
                if (AreEqual(target, any))
                {
                    value = source;
                }
                if (AreEqual(storedLinkValue.Source, value) ||
```

```
                        AreEqual(storedLinkValue.Target, value))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @continue;
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪  поддерживаются.");
        }

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

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

```
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;
```

```
            }
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink GetZero() => default;

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

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

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

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

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

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

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

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

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

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

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

        #region Disposable

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

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

        #endregion
    }
}
```

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

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

```

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class UnusedLinksListMethods<TLink> :
    ↪  AbsoluteCircularDoublyLinkedListMethods<TLink>, ILinksListMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪  UncheckedConverter<TLink, long>.Default;

        private readonly byte* _links;
        private readonly byte* _header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnusedLinksListMethods(byte* links, byte* header)
        {
            _links = links;
            _header = header;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLast() => GetHeaderReference().LastFreeLink;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize() => GetHeaderReference().FreeLinks;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
        ↪  element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
        ↪  element;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
    }
}
```

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

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

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

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

```

```
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;
21          public TLink LeftAsTarget;
22          public TLink RightAsTarget;
23          public TLink SizeAsTarget;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          public override bool Equals(object obj) => obj is RawLink<TLink> link ? Equals(link) :
            ↪  false;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public bool Equals(RawLink<TLink> other)
30              => _equalityComparer.Equals(Source, other.Source)
31              && _equalityComparer.Equals(Target, other.Target)
32              && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
33              && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
34              && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
35              && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
36              && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
37              && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public override int GetHashCode() => (Source, Target, LeftAsSource, RightAsSource,
            ↪  SizeAsSource, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          public static bool operator ==(RawLink<TLink> left, RawLink<TLink> right) =>
            ↪  left.Equals(right);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          public static bool operator !=(RawLink<TLink> left, RawLink<TLink> right) => !(left ==
            ↪  right);
47      }
48  }
```

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

```
1   using System.Runtime.CompilerServices;
2   using Platform.Data.Doublets.Memory.United.Generic;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Memory.United.Specific
7   {
8       public unsafe abstract class UInt32LinksRecursionlessSizeBalancedTreeMethodsBase :
        ↪  LinksRecursionlessSizeBalancedTreeMethodsBase<uint>
9       {
10          protected new readonly RawLink<uint>* Links;
11          protected new readonly LinksHeader<uint>* Header;
12
13          protected UInt32LinksRecursionlessSizeBalancedTreeMethodsBase(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
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override bool GreaterThanZero(uint value) => value > 0U;
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override bool GreaterThan(uint first, uint second) => first > second;
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;
37
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe abstract class UInt32LinksSizeBalancedTreeMethodsBase :
    ↪ LinksSizeBalancedTreeMethodsBase<uint>
    {
        protected new readonly RawLink<uint>* Links;
        protected new readonly LinksHeader<uint>* Header;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
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
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override bool GreaterThanZero(uint value) => value > 0U;
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override bool GreaterThan(uint first, uint second) => first > second;
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          protected override bool GreaterOrEqualThanZero(uint value) => true; // value >= 0 is
            ↪  always true for uint
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          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  }
```

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

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Memory.United.Specific
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

```

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

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

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Memory.United.Specific
6   {
7       public unsafe class UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods :
        ↪  UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
8       {
9           public UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
            ↪  constants, RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links,
            ↪  header) { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsTarget;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override ref uint GetRightReference(uint node) => ref
            ↪  Links[node].RightAsTarget;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override uint GetLeft(uint node) => Links[node].LeftAsTarget;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override uint GetRight(uint node) => Links[node].RightAsTarget;
22
```

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }

        /// <summary>
        /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
        ↪  минимальным шагом расширения базы данных.
        /// </summary>
        /// <param name="address">Полный пусть к файлу базы данных.</param>
        /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
        ↪  байтах.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
        ↪  FileMappedResizableDirectMemory(address, memoryReservationStep),
        ↪  memoryReservationStep) { }

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

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

```
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 }
```

1.99 ./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 }
```

1.100 ./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
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
104         protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
            ↪ storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
            ↪ value & 3) & 7UL);
105
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
108
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
111     }
112 }
```

1.101 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksRecursionlessSizeBalancedTreeMeth

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.United.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.United.Specific
7  {
8      public unsafe abstract class UInt64LinksRecursionlessSizeBalancedTreeMethodsBase :
       ↪ LinksRecursionlessSizeBalancedTreeMethodsBase<ulong>
9      {
10         protected new readonly RawLink<ulong>* Links;
11         protected new readonly LinksHeader<ulong>* Header;
12
13         protected UInt64LinksRecursionlessSizeBalancedTreeMethodsBase(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
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override ulong Decrement(ulong value) => --value;
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override ulong Add(ulong first, ulong second) => first + second;
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
63          protected override ulong Subtract(ulong first, ulong second) => first - second;
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong 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(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  }
```

1.102 ./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
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override ulong Decrement(ulong value) => --value;
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          protected override ulong Add(ulong first, ulong second) => first + second;
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          protected override ulong Subtract(ulong first, ulong second) => first - second;
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong 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(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  }
```

1.103 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Memory.United.Specific
6   {
7       public unsafe class UInt64LinksSourcesAvlBalancedTreeMethods :
        ↪  UInt64LinksAvlBalancedTreeMethodsBase
8       {
9           public UInt64LinksSourcesAvlBalancedTreeMethods(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;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
            ↪  left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
            ↪  right;
```

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

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

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

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

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Memory.United.Specific
6   {
7       public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
        ↪  UInt64LinksSizeBalancedTreeMethodsBase
8       {
9           public UInt64LinksSourcesSizeBalancedTreeMethods(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;
```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
    ↪  UInt64LinksAvlBalancedTreeMethodsBase
    {
        public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↪  { }

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.107 ./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;
        }
    }
}
```

1.108 ./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 = OUL;
            link.RightAsTarget = OUL;
            link.SizeAsTarget = OUL;
        }
    }
}
```

1.109 ./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>
    {
```

```
17          private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
18          private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
19          private LinksHeader<ulong>* _header;
20          private RawLink<ulong>* _links;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          public UInt64UnitedMemoryLinks(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 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
```

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetOne() => 1UL;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        private readonly IConverter<TSource> _numberToAddressConverter;

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

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

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

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

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

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

        private readonly IConverter<TTarget> _addressToNumberConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

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

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

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

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<Doublet<TLink>, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkToItsFrequencyNumberConveter(
            ILinks<TLink> links,
            IProperty<TLink, TLink> frequencyPropertyOperator,
            IConverter<TLink> unaryNumberToAddressConverter)
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(Doublet<TLink> doublet)
        {
            var links = _links;
            var link = links.SearchOrDefault(doublet.Source, doublet.Target);
            if (_equalityComparer.Equals(link, default))
            {
                throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
            }
            var frequency = _frequencyPropertyOperator.Get(link);
            if (_equalityComparer.Equals(frequency, default))
            {
                return default;
            }
            var frequencyNumber = links.GetSource(frequency);
            return _unaryNumberToAddressConverter.Convert(frequencyNumber);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<int, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink[] _unaryNumberPowersOf2;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
        {
            _unaryNumberPowersOf2 = new TLink[64];
            _unaryNumberPowersOf2[0] = one;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(int power)
        {
            Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
            ↪  - 1), nameof(power));
            if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
            {
                return _unaryNumberPowersOf2[power];
            }
            var previousPowerOf2 = Convert(power - 1);
```

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

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

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

```
            };
            var unary = unaryOne;
            var number = _one;
            for (var i = 1; i < 64; i++)
            {
                unary = links.GetOrCreate(unary, unary);
                number = Double(number);
                unaryToUInt64.Add(unary, number);
            }
            return unaryToUInt64;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink Double(TLink number) =>
        ↪ _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;
        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
        ↪ TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
        ↪ = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(TLink sourceNumber)
        {
            var links = _links;
            var nullConstant = links.Constants.Null;
            var source = sourceNumber;
            var target = nullConstant;
            if (!_equalityComparer.Equals(source, nullConstant))
            {
                while (true)
                {
                    if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
                    {
                        SetBit(ref target, powerOf2Index);
                        break;
                    }
                    else
                    {
                        powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
                        SetBit(ref target, powerOf2Index);
                        source = links.GetTarget(source);
                    }
                }
            }
            return target;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Dictionary<TLink, int>
        ↪ CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink>
        ↪ powerOf2ToUnaryNumberConverter)
        {
            var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
            for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
            {
                unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
```

```
            }
            return unaryNumberPowerOf2Indicies;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void SetBit(ref TLink target, int powerOf2Index) => target =
        ↪  Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
    }
}
```

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

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

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

namespace Platform.Data.Doublets.PropertyOperators
{
    public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>, IProperties<TLink, TLink,
    ↪  TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink GetValue(TLink @object, TLink property)
        {
            var links = _links;
            var objectProperty = links.SearchOrDefault(@object, property);
            if (_equalityComparer.Equals(objectProperty, default))
            {
                return default;
            }
            var constants = links.Constants;
            var any = constants.Any;
            var query = new Link<TLink>(any, objectProperty, any);
            var valueLink = links.SingleOrDefault(query);
            if (valueLink == null)
            {
                return default;
            }
            return links.GetTarget(valueLink[constants.IndexPart]);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void SetValue(TLink @object, TLink property, TLink value)
        {
            var links = _links;
            var objectProperty = links.GetOrCreate(@object, property);
            links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
            links.GetOrCreate(objectProperty, value);
        }
    }
}
```

1.119 ./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);
            }
        }
    }
}
```

1.120 ./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.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.Sequences.Converters
{
    public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Convert(IList<TLink> sequence)
        {
            var length = sequence.Count;
            if (length < 1)
            {
```

```
                return default;
            }
            if (length == 1)
            {
                return sequence[0];
            }
            // Make copy of next layer
            if (length > 2)
            {
                // TODO: Try to use stackalloc (which at the moment is not working with
                ↪  generics) but will be possible with Sigil
                var halvedSequence = new TLink[(length / 2) + (length % 2)];
                HalveSequence(halvedSequence, sequence, length);
                sequence = halvedSequence;
                length = halvedSequence.Length;
            }
            // Keep creating layer after layer
            while (length > 2)
            {
                HalveSequence(sequence, sequence, length);
                length = (length / 2) + (length % 2);
            }
            return _links.GetOrCreate(sequence[0], sequence[1]);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
        {
            var loopedLength = length - (length % 2);
            for (var i = 0; i < loopedLength; i += 2)
            {
                destination[i / 2] = _links.GetOrCreate(source[i], source[i + 1]);
            }
            if (length > loopedLength)
            {
                destination[length / 2] = source[length - 1];
            }
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections;
using Platform.Converters;
using Platform.Singletons;
using Platform.Numbers;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    /// <remarks>
    /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
    ↪  Links на этапе сжатия.
    ///     А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
    ↪  таком случае тип значения элемента массива может быть любым, как char так и ulong.
    ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
    ↪  пар, а так же разом выполнить замену.
    /// </remarks>
    public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        private static readonly LinksConstants<TLink> _constants =
        ↪  Default<LinksConstants<TLink>>.Instance;
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

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

        private readonly IConverter<IList<TLink>, TLink> _baseConverter;
        private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
        private readonly TLink _minFrequencyToCompress;
        private readonly bool _doInitialFrequenciesIncrement;
        private Doublet<TLink> _maxDoublet;
        private LinkFrequency<TLink> _maxDoubletData;
```

```
34
35          private struct HalfDoublet
36          {
37              public TLink Element;
38              public LinkFrequency<TLink> DoubletData;
39
40              [MethodImpl(MethodImplOptions.AggressiveInlining)]
41              public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
42              {
43                  Element = element;
44                  DoubletData = doubletData;
45              }
46
47              public override string ToString() => $"{Element}: ({DoubletData})";
48          }
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
52              : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
            ↪  doInitialFrequenciesIncrement)
56              : this(links, baseConverter, doubletFrequenciesCache, _one,
                ↪  doInitialFrequenciesIncrement) { }
57
58          [MethodImpl(MethodImplOptions.AggressiveInlining)]
59          public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
            ↪  minFrequencyToCompress, bool doInitialFrequenciesIncrement)
60              : base(links)
61          {
62              _baseConverter = baseConverter;
63              _doubletFrequenciesCache = doubletFrequenciesCache;
64              if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
65              {
66                  minFrequencyToCompress = _one;
67              }
68              _minFrequencyToCompress = minFrequencyToCompress;
69              _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
70              ResetMaxDoublet();
71          }
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          public override TLink Convert(IList<TLink> source) =>
            ↪  _baseConverter.Convert(Compress(source));
75
76          /// <remarks>
77          /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
78          /// Faster version (doublets' frequencies dictionary is not recreated).
79          /// </remarks>
80          [MethodImpl(MethodImplOptions.AggressiveInlining)]
81          private IList<TLink> Compress(IList<TLink> sequence)
82          {
83              if (sequence.IsNullOrEmpty())
84              {
85                  return null;
86              }
87              if (sequence.Count == 1)
88              {
89                  return sequence;
90              }
91              if (sequence.Count == 2)
92              {
93                  return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
94              }
95              // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
96              var copy = new HalfDoublet[sequence.Count];
97              Doublet<TLink> doublet = default;
98              for (var i = 1; i < sequence.Count; i++)
99              {
100                 doublet = new Doublet<TLink>(sequence[i - 1], sequence[i]);
101                 LinkFrequency<TLink> data;
102                 if (_doInitialFrequenciesIncrement)
103                 {
104                     data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
105                 }
```

```
106                 else
107                 {
108                     data = _doubletFrequenciesCache.GetFrequency(ref doublet);
109                     if (data == null)
110                     {
111                         throw new NotSupportedException("If you ask not to increment
                            ↪  frequencies, it is expected that all frequencies for the sequence
                            ↪  are prepared.");
112                     }
113                 }
114                 copy[i - 1].Element = sequence[i - 1];
115                 copy[i - 1].DoubletData = data;
116                 UpdateMaxDoublet(ref doublet, data);
117             }
118             copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
119             copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
120             if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
121             {
122                 var newLength = ReplaceDoublets(copy);
123                 sequence = new TLink[newLength];
124                 for (int i = 0; i < newLength; i++)
125                 {
126                     sequence[i] = copy[i].Element;
127                 }
128             }
129             return sequence;
130         }
131 
132         /// <remarks>
133         /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
134         /// </remarks>
135         [MethodImpl(MethodImplOptions.AggressiveInlining)]
136         private int ReplaceDoublets(HalfDoublet[] copy)
137         {
138             var oldLength = copy.Length;
139             var newLength = copy.Length;
140             while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
141             {
142                 var maxDoubletSource = _maxDoublet.Source;
143                 var maxDoubletTarget = _maxDoublet.Target;
144                 if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
145                 {
146                     _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
                            ↪  maxDoubletTarget);
147                 }
148                 var maxDoubletReplacementLink = _maxDoubletData.Link;
149                 oldLength--;
150                 var oldLengthMinusTwo = oldLength - 1;
151                 // Substitute all usages
152                 int w = 0, r = 0; // (r == read, w == write)
153                 for (; r < oldLength; r++)
154                 {
155                     if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                            ↪  _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
156                     {
157                         if (r > 0)
158                         {
159                             var previous = copy[w - 1].Element;
160                             copy[w - 1].DoubletData.DecrementFrequency();
161                             copy[w - 1].DoubletData =
                                    ↪  _doubletFrequenciesCache.IncrementFrequency(previous,
                                    ↪  maxDoubletReplacementLink);
162                         }
163                         if (r < oldLengthMinusTwo)
164                         {
165                             var next = copy[r + 2].Element;
166                             copy[r + 1].DoubletData.DecrementFrequency();
167                             copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma
                                    ↪  xDoubletReplacementLink,
                                    ↪  next);
168                         }
169                         copy[w++].Element = maxDoubletReplacementLink;
170                         r++;
171                         newLength--;
172                     }
173                     else
174                     {
175                         copy[w++] = copy[r];
```

```
                    }
                }
                if (w < newLength)
                {
                    copy[w] = copy[r];
                }
                oldLength = newLength;
                ResetMaxDoublet();
                UpdateMaxDoublet(copy, newLength);
            }
            return newLength;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ResetMaxDoublet()
        {
            _maxDoublet = new Doublet<TLink>();
            _maxDoubletData = new LinkFrequency<TLink>();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
        {
            Doublet<TLink> doublet = default;
            for (var i = 1; i < length; i++)
            {
                doublet = new Doublet<TLink>(copy[i - 1].Element, copy[i].Element);
                UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
        {
            var frequency = data.Frequency;
            var maxFrequency = _maxDoubletData.Frequency;
            //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
            ↪  (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
            ↪  compression string data (and gives collisions quickly) */ _maxDoublet.Source +
            ↪  _maxDoublet.Target)))
            if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
               (_comparer.Compare(maxFrequency, frequency) < 0 ||
                ↪  (_equalityComparer.Equals(maxFrequency, frequency) &&
                ↪  _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
                ↪  Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
                ↪  better stability and better compression on sequent data and even on rundom
                ↪  numbers data (but gives collisions anyway) */
            {
                _maxDoublet = doublet;
                _maxDoubletData = data;
            }
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    public abstract class LinksListToSequenceConverterBase<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<IList<TLink>, TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksListToSequenceConverterBase(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public abstract TLink Convert(IList<TLink> source);
    }
}
```

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

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

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
        ↪  sequenceToItsLocalElementLevelsConverter) : base(links)
            => _sequenceToItsLocalElementLevelsConverter =
            ↪  sequenceToItsLocalElementLevelsConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public OptimalVariantConverter(ILinks<TLink> links, LinkFrequenciesCache<TLink>
        ↪  linkFrequenciesCache)
            : this(links, new SequenceToItsLocalElementLevelsConverter<TLink>(links, new Frequen
            ↪  ciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>(linkFrequenciesCache))) {
            ↪  }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Convert(IList<TLink> sequence)
        {
            var length = sequence.Count;
            if (length == 1)
            {
                return sequence[0];
            }
            if (length == 2)
            {
                return _links.GetOrCreate(sequence[0], sequence[1]);
            }
            sequence = sequence.ToArray();
            var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
            while (length > 2)
            {
                var levelRepeat = 1;
                var currentLevel = levels[0];
                var previousLevel = levels[0];
                var skipOnce = false;
                var w = 0;
                for (var i = 1; i < length; i++)
                {
                    if (_equalityComparer.Equals(currentLevel, levels[i]))
                    {
                        levelRepeat++;
                        skipOnce = false;
                        if (levelRepeat == 2)
                        {
                            sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
                            var newLevel = i >= length - 1 ?
                                GetPreviousLowerThanCurrentOrCurrent(previousLevel,
                                ↪  currentLevel) :
                                i < 2 ?
                                GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
                                GetGreatestNeigbourLowerThanCurrentOrCurrent(previousLevel,
                                ↪  currentLevel, levels[i + 1]);
                            levels[w] = newLevel;
                            previousLevel = currentLevel;
                            w++;
                            levelRepeat = 0;
                            skipOnce = true;
                        }
                        else if (i == length - 1)
                        {
```

```
                            sequence[w] = sequence[i];
                            levels[w] = levels[i];
                            w++;
                        }
                    }
                    else
                    {
                        currentLevel = levels[i];
                        levelRepeat = 1;
                        if (skipOnce)
                        {
                            skipOnce = false;
                        }
                        else
                        {
                            sequence[w] = sequence[i - 1];
                            levels[w] = levels[i - 1];
                            previousLevel = levels[w];
                            w++;
                        }
                        if (i == length - 1)
                        {
                            sequence[w] = sequence[i];
                            levels[w] = levels[i];
                            w++;
                        }
                    }
                }
                length = w;
            }
            return _links.GetOrCreate(sequence[0], sequence[1]);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink
        ↪  current, TLink next)
        {
            return _comparer.Compare(previous, next) > 0
                ? _comparer.Compare(previous, current) < 0 ? previous : current
                : _comparer.Compare(next, current) < 0 ? next : current;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
        ↪  _comparer.Compare(next, current) < 0 ? next : current;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
        ↪  => _comparer.Compare(previous, current) < 0 ? previous : current;
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<IList<TLink>>
    {
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
        ↪  IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
        ↪  => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<TLink> Convert(IList<TLink> sequence)
        {
            var levels = new TLink[sequence.Count];
            levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
            for (var i = 1; i < sequence.Count - 1; i++)
```

```
            {
                var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
                var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
                levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
            }
            levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
            ↪  sequence[sequence.Count - 1]);
            return levels;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink GetFrequencyNumber(TLink source, TLink target) =>
        ↪  _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.CriterionMatchers
{
    public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
    ↪  ICriterionMatcher<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsMatched(TLink argument) => _links.IsPartialPoint(argument);
    }
}
```

1.126 ./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.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.Sequences.CriterionMatchers
{
    public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly ILinks<TLink> _links;
        private readonly TLink _sequenceMarkerLink;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
        {
            _links = links;
            _sequenceMarkerLink = sequenceMarkerLink;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsMatched(TLink sequenceCandidate)
            => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
            || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
            ↪  sequenceCandidate), _links.Constants.Null);
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;
using Platform.Data.Doublets.Sequences.HeightProviders;
using Platform.Data.Sequences;

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

namespace Platform.Data.Doublets.Sequences
{
    public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceAppender<TLink>
```

```
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly IStack<TLink> _stack;
        private readonly ISequenceHeightProvider<TLink> _heightProvider;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
        ↪  ISequenceHeightProvider<TLink> heightProvider)
            : base(links)
        {
            _stack = stack;
            _heightProvider = heightProvider;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Append(TLink sequence, TLink appendant)
        {
            var cursor = sequence;
            var links = _links;
            while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
            {
                var source = links.GetSource(cursor);
                var target = links.GetTarget(cursor);
                if (_equalityComparer.Equals(_heightProvider.Get(source),
                ↪  _heightProvider.Get(target)))
                {
                    break;
                }
                else
                {
                    _stack.Push(source);
                    cursor = target;
                }
            }
            var left = cursor;
            var right = appendant;
            while (!_equalityComparer.Equals(cursor = _stack.Pop(), links.Constants.Null))
            {
                right = links.GetOrCreate(left, right);
                left = cursor;
            }
            return links.GetOrCreate(left, right);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences
{
    public class DuplicateSegmentsCounter<TLink> : ICounter<int>
    {
        private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
        ↪  _duplicateFragmentsProvider;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
        ↪  IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
        ↪  duplicateFragmentsProvider;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
    }
}
```

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

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

```
using Platform.Collections.Lists;
using Platform.Collections.Segments;
using Platform.Collections.Segments.Walkers;
using Platform.Singletons;
using Platform.Converters;
using Platform.Data.Doublets.Unicode;

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

namespace Platform.Data.Doublets.Sequences
{
    public class DuplicateSegmentsProvider<TLink> :
    ↪ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
    ↪ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪ UncheckedConverter<TLink, long>.Default;
        private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
        ↪ UncheckedConverter<TLink, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
        ↪ UncheckedConverter<ulong, TLink>.Default;

        private readonly ILinks<TLink> _links;
        private readonly ILinks<TLink> _sequences;
        private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
        private BitString _visited;

        private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
        ↪ IList<TLink>>>
        {
            private readonly IListEqualityComparer<TLink> _listComparer;

            public ItemEquilityComparer() => _listComparer =
            ↪ Default<IListEqualityComparer<TLink>>.Instance;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
            ↪ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
            ↪ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
            ↪ right.Value);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
            ↪ (_listComparer.GetHashCode(pair.Key),
            ↪ _listComparer.GetHashCode(pair.Value)).GetHashCode();
        }

        private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
        {
            private readonly IListComparer<TLink> _listComparer;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
            ↪ KeyValuePair<IList<TLink>, IList<TLink>> right)
            {
                var intermediateResult = _listComparer.Compare(left.Key, right.Key);
                if (intermediateResult == 0)
                {
                    intermediateResult = _listComparer.Compare(left.Value, right.Value);
                }
                return intermediateResult;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
            : base(minimumStringSegmentLength: 2)
        {
            _links = links;
            _sequences = sequences;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
        {
            _groups = new HashSet<KeyValuePair<IList<TLink>,
            ↪ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
```

```
73              var links = _links;
74              var count = links.Count();
75              _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
76              links.Each(link =>
77              {
78                  var linkIndex = links.GetIndex(link);
79                  var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
80                  var constants = links.Constants;
81                  if (!_visited.Get(linkBitIndex))
82                  {
83                      var sequenceElements = new List<TLink>();
84                      var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
85                      _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
                        ↪  LinkAddress<TLink>(linkIndex));
86                      if (sequenceElements.Count > 2)
87                      {
88                          WalkAll(sequenceElements);
89                      }
90                  }
91                  return constants.Continue;
92              });
93              var resultList = _groups.ToList();
94              var comparer = Default<ItemComparer>.Instance;
95              resultList.Sort(comparer);
96  #if DEBUG
97              foreach (var item in resultList)
98              {
99                  PrintDuplicates(item);
100             }
101 #endif
102             return resultList;
103         }
104
105         [MethodImpl(MethodImplOptions.AggressiveInlining)]
106         protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
            ↪  length) => new Segment<TLink>(elements, offset, length);
107
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         protected override void OnDublicateFound(Segment<TLink> segment)
110         {
111             var duplicates = CollectDuplicatesForSegment(segment);
112             if (duplicates.Count > 1)
113             {
114                 _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                    ↪  duplicates));
115             }
116         }
117
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
119         private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
120         {
121             var duplicates = new List<TLink>();
122             var readAsElement = new HashSet<TLink>();
123             var restrictions = segment.ShiftRight();
124             var constants = _links.Constants;
125             restrictions[0] = constants.Any;
126             _sequences.Each(sequence =>
127             {
128                 var sequenceIndex = sequence[constants.IndexPart];
129                 duplicates.Add(sequenceIndex);
130                 readAsElement.Add(sequenceIndex);
131                 return constants.Continue;
132             }, restrictions);
133             if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
134             {
135                 return new List<TLink>();
136             }
137             foreach (var duplicate in duplicates)
138             {
139                 var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
140                 _visited.Set(duplicateBitIndex);
141             }
142             if (_sequences is Sequences sequencesExperiments)
143             {
144                 var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
                    ↪  ashSet<ulong>)(object)readAsElement,
                    ↪  (IList<ulong>)segment);
145                 foreach (var partiallyMatchedSequence in partiallyMatched)
```

```
                {
                    var sequenceIndex =
                    ↪  _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
                    duplicates.Add(sequenceIndex);
                }
            }
            duplicates.Sort();
            return duplicates;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
        {
            if (!(_links is ILinks<ulong> ulongLinks))
            {
                return;
            }
            var duplicatesKey = duplicatesItem.Key;
            var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
            Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
            var duplicatesList = duplicatesItem.Value;
            for (int i = 0; i < duplicatesList.Count; i++)
            {
                var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
                var formatedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
                ↪  Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
                ↪  UnicodeMap.IsCharLink(link.Index) ?
                ↪  sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
                Console.WriteLine(formatedSequenceStructure);
                var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
                ↪  ulongLinks);
                Console.WriteLine(sequenceString);
            }
            Console.WriteLine();
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    /// <remarks>
    /// Can be used to operate with many CompressingConverters (to keep global frequencies data
    ↪  between them).
    /// TODO: Extract interface to implement frequencies storage inside Links storage
    /// </remarks>
    public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

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

        private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
        private readonly ICounter<TLink, TLink> _frequencyCounter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
            : base(links)
        {
            _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
            ↪  DoubletComparer<TLink>.Default);
            _frequencyCounter = frequencyCounter;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
        {
            var doublet = new Doublet<TLink>(source, target);
```

```
            return GetFrequency(ref doublet);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
        {
            _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
            return data;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncrementFrequencies(IList<TLink> sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                IncrementFrequency(sequence[i - 1], sequence[i]);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
        {
            var doublet = new Doublet<TLink>(source, target);
            return IncrementFrequency(ref doublet);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void PrintFrequencies(IList<TLink> sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                PrintFrequency(sequence[i - 1], sequence[i]);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void PrintFrequency(TLink source, TLink target)
        {
            var number = GetFrequency(source, target).Frequency;
            Console.WriteLine("({0},{1}) - {2}", source, target, number);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
        {
            if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
            {
                data.IncrementFrequency();
            }
            else
            {
                var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
                data = new LinkFrequency<TLink>(_one, link);
                if (!_equalityComparer.Equals(link, default))
                {
                    data.Frequency = Arithmetic.Add(data.Frequency,
                    ↪  _frequencyCounter.Count(link));
                }
                _doubletsCache.Add(doublet, data);
            }
            return data;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ValidateFrequencies()
        {
            foreach (var entry in _doubletsCache)
            {
                var value = entry.Value;
                var linkIndex = value.Link;
                if (!_equalityComparer.Equals(linkIndex, default))
                {
                    var frequency = value.Frequency;
                    var count = _frequencyCounter.Count(linkIndex);
                    // TODO: Why `frequency` always greater than `count` by 1?
                    if (((_comparer.Compare(frequency, count) > 0) &&
                    ↪  (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
                     || ((_comparer.Compare(count, frequency) > 0) &&
                      ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
```

```
                    {
                        throw new InvalidOperationException("Frequencies validation failed.");
                    }
                }
                //else
                //{
                //    if (value.Frequency > 0)
                //    {
                //        var frequency = value.Frequency;
                //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
                //        var count = _countLinkFrequency(linkIndex);

                //        if ((frequency > count && frequency - count > 1) || (count > frequency
                ↪  && count - frequency > 1))
                //            throw new InvalidOperationException("Frequencies validation
                ↪  failed.");
                //    }
                //}
            }
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    public class LinkFrequency<TLink>
    {
        public TLink Frequency { get; set; }
        public TLink Link { get; set; }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency(TLink frequency, TLink link)
        {
            Frequency = frequency;
            Link = link;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency() { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => $"F: {Frequency}, L: {Link}";
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
    ↪  IConverter<Doublet<TLink>, TLink>
    {
        private readonly LinkFrequenciesCache<TLink> _cache;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
        ↪  FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
        ↪  cache) => _cache = cache;

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

1.133 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneO

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
    ↪  SequenceSymbolFrequencyOneOffCounter<TLink>
    {
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
        ↪  ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
            : base(links, sequenceLink, symbol)
            => _markedSequenceMatcher = markedSequenceMatcher;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Count()
        {
            if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
            {
                return default;
            }
            return base.Count();
        }
    }
}
```

1.134 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCoun

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Numbers;
using Platform.Data.Sequences;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        protected readonly ILinks<TLink> _links;
        protected readonly TLink _sequenceLink;
        protected readonly TLink _symbol;
        protected TLink _total;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
        ↪  TLink symbol)
        {
            _links = links;
            _sequenceLink = sequenceLink;
            _symbol = symbol;
            _total = default;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Count()
        {
            if (_comparer.Compare(_total, default) > 0)
            {
                return _total;
            }
            StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
            ↪  IsElement, VisitElement);
            return _total;
        }

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

```

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

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

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

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

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

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

```
1   using System.Runtime.CompilerServices;
2   using Platform.Interfaces;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7   {
```

```
    public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
    {
        private readonly ILinks<TLink> _links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Count(TLink symbol) => new
        ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        protected readonly ILinks<TLink> _links;
        protected readonly TLink _symbol;
        protected readonly HashSet<TLink> _visits;
        protected TLink _total;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
        {
            _links = links;
            _symbol = symbol;
            _visits = new HashSet<TLink>();
            _total = default;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Count()
        {
            if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
            {
                return _total;
            }
            CountCore(_symbol);
            return _total;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void CountCore(TLink link)
        {
            var any = _links.Constants.Any;
            if (_equalityComparer.Equals(_links.Count(any, link), default))
            {
                CountSequenceSymbolFrequency(link);
            }
            else
            {
                _links.Each(EachElementHandler, any, link);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void CountSequenceSymbolFrequency(TLink link)
        {
            var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
            ↪  link, _symbol);
            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink EachElementHandler(IList<TLink> doublet)
        {
            var constants = _links.Constants;
```

```
            var doubletIndex = doublet[constants.IndexPart];
            if (_visits.Add(doubletIndex))
            {
                CountCore(doubletIndex);
            }
            return constants.Continue;
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    public class CachedSequenceHeightProvider<TLink> : ISequenceHeightProvider<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _heightPropertyMarker;
        private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
        private readonly IConverter<TLink> _addressToUnaryNumberConverter;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;
        private readonly IProperties<TLink, TLink, TLink> _propertyOperator;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachedSequenceHeightProvider(
            ISequenceHeightProvider<TLink> baseHeightProvider,
            IConverter<TLink> addressToUnaryNumberConverter,
            IConverter<TLink> unaryNumberToAddressConverter,
            TLink heightPropertyMarker,
            IProperties<TLink, TLink, TLink> propertyOperator)
        {
            _heightPropertyMarker = heightPropertyMarker;
            _baseHeightProvider = baseHeightProvider;
            _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
            _propertyOperator = propertyOperator;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Get(TLink sequence)
        {
            TLink height;
            var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
            if (_equalityComparer.Equals(heightValue, default))
            {
                height = _baseHeightProvider.Get(sequence);
                heightValue = _addressToUnaryNumberConverter.Convert(height);
                _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
            }
            else
            {
                height = _unaryNumberToAddressConverter.Convert(heightValue);
            }
            return height;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceHeightProvider<TLink>
    {
        private readonly ICriterionMatcher<TLink> _elementMatcher;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
14          public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
            ↪  elementMatcher) : base(links) => _elementMatcher = elementMatcher;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public TLink Get(TLink sequence)
18          {
19              var height = default(TLink);
20              var pairOrElement = sequence;
21              while (!_elementMatcher.IsMatched(pairOrElement))
22              {
23                  pairOrElement = _links.GetTarget(pairOrElement);
24                  height = Arithmetic.Increment(height);
25              }
26              return height;
27          }
28      }
29  }
```

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

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

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Sequences.Indexes
8   {
9       public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
10      {
11          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
12
13          private readonly LinkFrequenciesCache<TLink> _cache;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
            ↪  _cache = cache;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          public bool Add(IList<TLink> sequence)
20          {
21              var indexed = true;
22              var i = sequence.Count;
23              while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
                ↪  { }
24              for (; i >= 1; i--)
25              {
26                  _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
27              }
28              return indexed;
29          }
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          private bool IsIndexedWithIncrement(TLink source, TLink target)
33          {
34              var frequency = _cache.GetFrequency(source, target);
35              if (frequency == null)
36              {
37                  return false;
38              }
39              var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
40              if (indexed)
41              {
42                  _cache.IncrementFrequency(source, target);
43              }
44              return indexed;
45          }
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool MightContain(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
            return indexed;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsIndexed(TLink source, TLink target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            return !_equalityComparer.Equals(frequency.Frequency, default);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
    ↪  ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
        private readonly IIncrementer<TLink> _frequencyIncrementer;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IProperty<TLink, TLink>
        ↪  frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _frequencyIncrementer = frequencyIncrementer;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
            ↪  { }
            for (; i >= 1; i--)
            {
                Increment(_links.GetOrCreate(sequence[i - 1], sequence[i]));
            }
            return indexed;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsIndexedWithIncrement(TLink source, TLink target)
        {
            var link = _links.SearchOrDefault(source, target);
            var indexed = !_equalityComparer.Equals(link, default);
            if (indexed)
            {
                Increment(link);
            }
            return indexed;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void Increment(TLink link)
        {
```

```
            var previousFrequency = _frequencyPropertyOperator.Get(link);
            var frequency = _frequencyIncrementer.Increment(previousFrequency);
            _frequencyPropertyOperator.Set(link, frequency);
        }
    }
}
```

1.144 ./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.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.Sequences.Indexes
{
    public interface ISequenceIndex<TLink>
    {
        /// <summary>
        /// Индексирует последовательность глобально, и возвращает значение,
        /// определяющие была ли запрошенная последовательность проиндексирована ранее.
        /// </summary>
        /// <param name="sequence">Последовательность для индексации.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        bool Add(IList<TLink> sequence);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        bool MightContain(IList<TLink> sequence);
    }
}
```

1.145 ./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.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.Sequences.Indexes
{
    public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed =
            ↪  !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪  default))) { }
            for (; i >= 1; i--)
            {
                _links.GetOrCreate(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool MightContain(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed =
            ↪  !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪  default))) { }
            return indexed;
        }
    }
}
```

1.146 ./csharp/Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.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.Sequences.Indexes
{
    public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly ISynchronizedLinks<TLink> _links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            var links = _links.Unsync;
            _links.SyncRoot.ExecuteReadOperation(() =>
            {
                while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪  sequence[i]), default))) { }
            });
            if (!indexed)
            {
                _links.SyncRoot.ExecuteWriteOperation(() =>
                {
                    for (; i >= 1; i--)
                    {
                        links.GetOrCreate(sequence[i - 1], sequence[i]);
                    }
                });
            }
            return indexed;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool MightContain(IList<TLink> sequence)
        {
            var links = _links.Unsync;
            return _links.SyncRoot.ExecuteReadOperation(() =>
            {
                var indexed = true;
                var i = sequence.Count;
                while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪  sequence[i]), default))) { }
                return indexed;
            });
        }
    }
}
```

1.147 ./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.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.Sequences.Indexes
{
    public class Unindex<TLink> : ISequenceIndex<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Add(IList<TLink> sequence) => false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool MightContain(IList<TLink> sequence) => true;
    }
}
```

1.148 ./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs

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

```
using Platform.Collections;
using Platform.Collections.Sets;
using Platform.Collections.Stacks;
using Platform.Data.Exceptions;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Walkers;
using LinkIndex = System.UInt64;
using Stack = System.Collections.Generic.Stack<ulong>;

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

namespace Platform.Data.Doublets.Sequences
{
    partial class Sequences
    {
        #region Create All Variants (Not Practical)

        /// <remarks>
        /// Number of links that is needed to generate all variants for
        /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ulong[] CreateAllVariants2(ulong[] sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return Array.Empty<ulong>();
                }
                Links.EnsureLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return sequence;
                }
                return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
        {
#if DEBUG
            if ((stopAt - startAt) < 0)
            {
                throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
                ↪  меньше или равен stopAt");
            }
#endif
            if ((stopAt - startAt) == 0)
            {
                return new[] { sequence[startAt] };
            }
            if ((stopAt - startAt) == 1)
            {
                return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
            }
            var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
            var last = 0;
            for (var splitter = startAt; splitter < stopAt; splitter++)
            {
                var left = CreateAllVariants2Core(sequence, startAt, splitter);
                var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
                for (var i = 0; i < left.Length; i++)
                {
                    for (var j = 0; j < right.Length; j++)
                    {
                        var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
                        if (variant == Constants.Null)
                        {
                            throw new NotImplementedException("Creation cancellation is not
                            ↪  implemented.");
                        }
                        variants[last++] = variant;
                    }
                }
            }
            return variants;
```

```
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> CreateAllVariants1(params ulong[] sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return new List<ulong>();
                }
                Links.Unsync.EnsureLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return new List<ulong> { sequence[0] };
                }
                var results = new
                ↪  List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
                return CreateAllVariants1Core(sequence, results);
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
        {
            if (sequence.Length == 2)
            {
                var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
                if (link == Constants.Null)
                {
                    throw new NotImplementedException("Creation cancellation is not
                    ↪  implemented.");
                }
                results.Add(link);
                return results;
            }
            var innerSequenceLength = sequence.Length - 1;
            var innerSequence = new ulong[innerSequenceLength];
            for (var li = 0; li < innerSequenceLength; li++)
            {
                var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
                if (link == Constants.Null)
                {
                    throw new NotImplementedException("Creation cancellation is not
                    ↪  implemented.");
                }
                for (var isi = 0; isi < li; isi++)
                {
                    innerSequence[isi] = sequence[isi];
                }
                innerSequence[li] = link;
                for (var isi = li + 1; isi < innerSequenceLength; isi++)
                {
                    innerSequence[isi] = sequence[isi + 1];
                }
                CreateAllVariants1Core(innerSequence, results);
            }
            return results;
        }

        #endregion

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> Each1(params ulong[] sequence)
        {
            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
            Each1(link =>
            {
                if (!visitedLinks.Contains(link))
                {
                    visitedLinks.Add(link); // изучить почему случаются повторы
                }
                return true;
            }, sequence);
            return visitedLinks;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
158         private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
159         {
160             if (sequence.Length == 2)
161             {
162                 Links.Unsync.Each(sequence[0], sequence[1], handler);
163             }
164             else
165             {
166                 var innerSequenceLength = sequence.Length - 1;
167                 for (var li = 0; li < innerSequenceLength; li++)
168                 {
169                     var left = sequence[li];
170                     var right = sequence[li + 1];
171                     if (left == 0 && right == 0)
172                     {
173                         continue;
174                     }
175                     var linkIndex = li;
176                     ulong[] innerSequence = null;
177                     Links.Unsync.Each(doublet =>
178                     {
179                         if (innerSequence == null)
180                         {
181                             innerSequence = new ulong[innerSequenceLength];
182                             for (var isi = 0; isi < linkIndex; isi++)
183                             {
184                                 innerSequence[isi] = sequence[isi];
185                             }
186                             for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
187                             {
188                                 innerSequence[isi] = sequence[isi + 1];
189                             }
190                         }
191                         innerSequence[linkIndex] = doublet[Constants.IndexPart];
192                         Each1(handler, innerSequence);
193                         return Constants.Continue;
194                     }, Constants.Any, left, right);
195                 }
196             }
197         }
198 
199         [MethodImpl(MethodImplOptions.AggressiveInlining)]
200         public HashSet<ulong> EachPart(params ulong[] sequence)
201         {
202             var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
203             EachPartCore(link =>
204             {
205                 var linkIndex = link[Constants.IndexPart];
206                 if (!visitedLinks.Contains(linkIndex))
207                 {
208                     visitedLinks.Add(linkIndex); // изучить почему случаются повторы
209                 }
210                 return Constants.Continue;
211             }, sequence);
212             return visitedLinks;
213         }
214 
215         [MethodImpl(MethodImplOptions.AggressiveInlining)]
216         public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
217         {
218             var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
219             EachPartCore(link =>
220             {
221                 var linkIndex = link[Constants.IndexPart];
222                 if (!visitedLinks.Contains(linkIndex))
223                 {
224                     visitedLinks.Add(linkIndex); // изучить почему случаются повторы
225                     return handler(new LinkAddress<LinkIndex>(linkIndex));
226                 }
227                 return Constants.Continue;
228             }, sequence);
229         }
230 
231         [MethodImpl(MethodImplOptions.AggressiveInlining)]
232         private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
            ↪ sequence)
233         {
234             if (sequence.IsNullOrEmpty())
```

```
235             {
236                 return;
237             }
238             Links.EnsureLinkIsAnyOrExists(sequence);
239             if (sequence.Length == 1)
240             {
241                 var link = sequence[0];
242                 if (link > 0)
243                 {
244                     handler(new LinkAddress<LinkIndex>(link));
245                 }
246                 else
247                 {
248                     Links.Each(Constants.Any, Constants.Any, handler);
249                 }
250             }
251             else if (sequence.Length == 2)
252             {
253                 //_links.Each(sequence[0], sequence[1], handler);
254                 //  o_|      x_o ...
255                 // x_|        |___|
256                 Links.Each(sequence[1], Constants.Any, doublet =>
257                 {
258                     var match = Links.SearchOrDefault(sequence[0], doublet);
259                     if (match != Constants.Null)
260                     {
261                         handler(new LinkAddress<LinkIndex>(match));
262                     }
263                     return true;
264                 });
265                 // |_x      ... x_o
266                 //  |_o      |___|
267                 Links.Each(Constants.Any, sequence[0], doublet =>
268                 {
269                     var match = Links.SearchOrDefault(doublet, sequence[1]);
270                     if (match != 0)
271                     {
272                         handler(new LinkAddress<LinkIndex>(match));
273                     }
274                     return true;
275                 });
276                 //           ._x o_.
277                 //            |___|
278                 PartialStepRight(x => handler(x), sequence[0], sequence[1]);
279             }
280             else
281             {
282                 throw new NotImplementedException();
283             }
284         }
285 
286         [MethodImpl(MethodImplOptions.AggressiveInlining)]
287         private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
288         {
289             Links.Unsync.Each(Constants.Any, left, doublet =>
290             {
291                 StepRight(handler, doublet, right);
292                 if (left != doublet)
293                 {
294                     PartialStepRight(handler, doublet, right);
295                 }
296                 return true;
297             });
298         }
299 
300         [MethodImpl(MethodImplOptions.AggressiveInlining)]
301         private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
302         {
303             Links.Unsync.Each(left, Constants.Any, rightStep =>
304             {
305                 TryStepRightUp(handler, right, rightStep);
306                 return true;
307             });
308         }
309 
310         [MethodImpl(MethodImplOptions.AggressiveInlining)]
311         private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
            ↪  stepFrom)
```

```
312         {
313             var upStep = stepFrom;
314             var firstSource = Links.Unsync.GetTarget(upStep);
315             while (firstSource != right && firstSource != upStep)
316             {
317                 upStep = firstSource;
318                 firstSource = Links.Unsync.GetSource(upStep);
319             }
320             if (firstSource == right)
321             {
322                 handler(new LinkAddress<LinkIndex>(stepFrom));
323             }
324         }
325 
326         // TODO: Test
327         [MethodImpl(MethodImplOptions.AggressiveInlining)]
328         private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
329         {
330             Links.Unsync.Each(right, Constants.Any, doublet =>
331             {
332                 StepLeft(handler, left, doublet);
333                 if (right != doublet)
334                 {
335                     PartialStepLeft(handler, left, doublet);
336                 }
337                 return true;
338             });
339         }
340 
341         [MethodImpl(MethodImplOptions.AggressiveInlining)]
342         private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
343         {
344             Links.Unsync.Each(Constants.Any, right, leftStep =>
345             {
346                 TryStepLeftUp(handler, left, leftStep);
347                 return true;
348             });
349         }
350 
351         [MethodImpl(MethodImplOptions.AggressiveInlining)]
352         private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
353         {
354             var upStep = stepFrom;
355             var firstTarget = Links.Unsync.GetSource(upStep);
356             while (firstTarget != left && firstTarget != upStep)
357             {
358                 upStep = firstTarget;
359                 firstTarget = Links.Unsync.GetTarget(upStep);
360             }
361             if (firstTarget == left)
362             {
363                 handler(new LinkAddress<LinkIndex>(stepFrom));
364             }
365         }
366 
367         [MethodImpl(MethodImplOptions.AggressiveInlining)]
368         private bool StartsWith(ulong sequence, ulong link)
369         {
370             var upStep = sequence;
371             var firstSource = Links.Unsync.GetSource(upStep);
372             while (firstSource != link && firstSource != upStep)
373             {
374                 upStep = firstSource;
375                 firstSource = Links.Unsync.GetSource(upStep);
376             }
377             return firstSource == link;
378         }
379 
380         [MethodImpl(MethodImplOptions.AggressiveInlining)]
381         private bool EndsWith(ulong sequence, ulong link)
382         {
383             var upStep = sequence;
384             var lastTarget = Links.Unsync.GetTarget(upStep);
385             while (lastTarget != link && lastTarget != upStep)
386             {
387                 upStep = lastTarget;
388                 lastTarget = Links.Unsync.GetTarget(upStep);
389             }
390             return lastTarget == link;
```

```
391         }
392
393         [MethodImpl(MethodImplOptions.AggressiveInlining)]
394         public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
395         {
396             return _sync.ExecuteReadOperation(() =>
397             {
398                 var results = new List<ulong>();
399                 if (sequence.Length > 0)
400                 {
401                     Links.EnsureLinkExists(sequence);
402                     var firstElement = sequence[0];
403                     if (sequence.Length == 1)
404                     {
405                         results.Add(firstElement);
406                         return results;
407                     }
408                     if (sequence.Length == 2)
409                     {
410                         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
411                         if (doublet != Constants.Null)
412                         {
413                             results.Add(doublet);
414                         }
415                         return results;
416                     }
417                     var linksInSequence = new HashSet<ulong>(sequence);
418                     void handler(IList<LinkIndex> result)
419                     {
420                         var resultIndex = result[Links.Constants.IndexPart];
421                         var filterPosition = 0;
422                         StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
                            ↪  Links.Unsync.GetTarget,
423                             x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                                ↪  x =>
424                             {
425                                 if (filterPosition == sequence.Length)
426                                 {
427                                     filterPosition = -2; // Длиннее чем нужно
428                                     return false;
429                                 }
430                                 if (x != sequence[filterPosition])
431                                 {
432                                     filterPosition = -1;
433                                     return false; // Начинается иначе
434                                 }
435                                 filterPosition++;
436
437                                 return true;
438                             });
439                         if (filterPosition == sequence.Length)
440                         {
441                             results.Add(resultIndex);
442                         }
443                     }
444                     if (sequence.Length >= 2)
445                     {
446                         StepRight(handler, sequence[0], sequence[1]);
447                     }
448                     var last = sequence.Length - 2;
449                     for (var i = 1; i < last; i++)
450                     {
451                         PartialStepRight(handler, sequence[i], sequence[i + 1]);
452                     }
453                     if (sequence.Length >= 3)
454                     {
455                         StepLeft(handler, sequence[sequence.Length - 2],
                            ↪  sequence[sequence.Length - 1]);
456                     }
457                 }
458                 return results;
459             });
460         }
461
462         [MethodImpl(MethodImplOptions.AggressiveInlining)]
463         public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
464         {
465             return _sync.ExecuteReadOperation(() =>
466             {
```

```
                var results = new HashSet<ulong>();
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    var firstElement = sequence[0];
                    if (sequence.Length == 1)
                    {
                        results.Add(firstElement);
                        return results;
                    }
                    if (sequence.Length == 2)
                    {
                        var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
                        if (doublet != Constants.Null)
                        {
                            results.Add(doublet);
                        }
                        return results;
                    }
                    var matcher = new Matcher(this, sequence, results, null);
                    if (sequence.Length >= 2)
                    {
                        StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
                    }
                    var last = sequence.Length - 2;
                    for (var i = 1; i < last; i++)
                    {
                        PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
                        ↪  sequence[i + 1]);
                    }
                    if (sequence.Length >= 3)
                    {
                        StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
                        ↪  sequence[sequence.Length - 1]);
                    }
                }
                return results;
            });
        }

        public const int MaxSequenceFormatSize = 200;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
        ↪  => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
        ↪  elementToString, bool insertComma, params LinkIndex[] knownElements) =>
        ↪  Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
        ↪  elementToString, insertComma, knownElements));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
        ↪  Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
        ↪  LinkIndex[] knownElements)
        {
            var linksInSequence = new HashSet<ulong>(knownElements);
            //var entered = new HashSet<ulong>();
            var sb = new StringBuilder();
            sb.Append('{');
            if (links.Exists(sequenceLink))
            {
                StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
                    x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
                    ↪  entered.AddAndReturnVoid, x => { }, entered.DoNotContains
                    {
                        if (insertComma && sb.Length > 1)
                        {
                            sb.Append(',');
                        }
                        //if (entered.Contains(element))
                        //{
                        //    sb.Append('{');
                        //    elementToString(sb, element);
                        //    sb.Append('}');
                        //}
                        //else
```

```
                    elementToString(sb, element);
                    if (sb.Length < MaxSequenceFormatSize)
                    {
                        return true;
                    }
                    sb.Append(insertComma ? ", ..." : "...");
                    return false;
                });
        }
        sb.Append('}');
        return sb.ToString();
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
    ↪  knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
    ↪  knownElements);

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
    ↪  LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↪  Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
    ↪  sequenceLink, elementToString, insertComma, knownElements));

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↪  Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    ↪  LinkIndex[] knownElements)
    {
        var linksInSequence = new HashSet<ulong>(knownElements);
        var entered = new HashSet<ulong>();
        var sb = new StringBuilder();
        sb.Append('{');
        if (links.Exists(sequenceLink))
        {
            StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
                x => linksInSequence.Contains(x) || links.IsFullPoint(x),
                ↪  entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
                {
                    if (insertComma && sb.Length > 1)
                    {
                        sb.Append(',');
                    }
                    if (entered.Contains(element))
                    {
                        sb.Append('{');
                        elementToString(sb, element);
                        sb.Append('}');
                    }
                    else
                    {
                        elementToString(sb, element);
                    }
                    if (sb.Length < MaxSequenceFormatSize)
                    {
                        return true;
                    }
                    sb.Append(insertComma ? ", ..." : "...");
                    return false;
                });
        }
        sb.Append('}');
        return sb.ToString();
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            if (sequence.Length > 0)
            {
                Links.EnsureLinkExists(sequence);
                var results = new HashSet<ulong>();
                for (var i = 0; i < sequence.Length; i++)
                {
                    AllUsagesCore(sequence[i], results);
                }
```

```
606                     var filteredResults = new List<ulong>();
607                     var linksInSequence = new HashSet<ulong>(sequence);
608                     foreach (var result in results)
609                     {
610                         var filterPosition = -1;
611                         StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
                            ↪ Links.Unsync.GetTarget,
612                             x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                                ↪ x =>
613                             {
614                                 if (filterPosition == (sequence.Length - 1))
615                                 {
616                                     return false;
617                                 }
618                                 if (filterPosition >= 0)
619                                 {
620                                     if (x == sequence[filterPosition + 1])
621                                     {
622                                         filterPosition++;
623                                     }
624                                     else
625                                     {
626                                         return false;
627                                     }
628                                 }
629                                 if (filterPosition < 0)
630                                 {
631                                     if (x == sequence[0])
632                                     {
633                                         filterPosition = 0;
634                                     }
635                                 }
636                                 return true;
637                             });
638                         if (filterPosition == (sequence.Length - 1))
639                         {
640                             filteredResults.Add(result);
641                         }
642                     }
643                     return filteredResults;
644                 }
645                 return new List<ulong>();
646             });
647         }
648 
649         [MethodImpl(MethodImplOptions.AggressiveInlining)]
650         public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
651         {
652             return _sync.ExecuteReadOperation(() =>
653             {
654                 if (sequence.Length > 0)
655                 {
656                     Links.EnsureLinkExists(sequence);
657                     var results = new HashSet<ulong>();
658                     for (var i = 0; i < sequence.Length; i++)
659                     {
660                         AllUsagesCore(sequence[i], results);
661                     }
662                     var filteredResults = new HashSet<ulong>();
663                     var matcher = new Matcher(this, sequence, filteredResults, null);
664                     matcher.AddAllPartialMatchedToResults(results);
665                     return filteredResults;
666                 }
667                 return new HashSet<ulong>();
668             });
669         }
670 
671         [MethodImpl(MethodImplOptions.AggressiveInlining)]
672         public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
            ↪ params ulong[] sequence)
673         {
674             return _sync.ExecuteReadOperation(() =>
675             {
676                 if (sequence.Length > 0)
677                 {
678                     Links.EnsureLinkExists(sequence);
679 
680                     var results = new HashSet<ulong>();
681                     var filteredResults = new HashSet<ulong>();
```

```
                    var matcher = new Matcher(this, sequence, filteredResults, handler);
                    for (var i = 0; i < sequence.Length; i++)
                    {
                        if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
                        {
                            return false;
                        }
                    }
                    return true;
                }
                return true;
            });
        }

        //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
        //{
        //    return Sync.ExecuteReadOperation(() =>
        //    {
        //        if (sequence.Length > 0)
        //        {
        //            _links.EnsureEachLinkIsAnyOrExists(sequence);

        //            var firstResults = new HashSet<ulong>();
        //            var lastResults = new HashSet<ulong>();

        //            var first = sequence.First(x => x != LinksConstants.Any);
        //            var last = sequence.Last(x => x != LinksConstants.Any);

        //            AllUsagesCore(first, firstResults);
        //            AllUsagesCore(last, lastResults);

        //            firstResults.IntersectWith(lastResults);

        //            //for (var i = 0; i < sequence.Length; i++)
        //            //    AllUsagesCore(sequence[i], results);

        //            var filteredResults = new HashSet<ulong>();
        //            var matcher = new Matcher(this, sequence, filteredResults, null);
        //            matcher.AddAllPartialMatchedToResults(firstResults);
        //            return filteredResults;
        //        }

        //        return new HashSet<ulong>();
        //    });
        //}

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation((Func<HashSet<ulong>>)(() =>
            {
                if (sequence.Length > 0)
                {
                    ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
                    ↪  (IList<ulong>)sequence);
                    var firstResults = new HashSet<ulong>();
                    var lastResults = new HashSet<ulong>();
                    var first = sequence.First(x => x != Constants.Any);
                    var last = sequence.Last(x => x != Constants.Any);
                    AllUsagesCore(first, firstResults);
                    AllUsagesCore(last, lastResults);
                    firstResults.IntersectWith(lastResults);
                    //for (var i = 0; i < sequence.Length; i++)
                    //    AllUsagesCore(sequence[i], results);
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new Matcher(this, sequence, filteredResults, null);
                    matcher.AddAllPartialMatchedToResults(firstResults);
                    return filteredResults;
                }
                return new HashSet<ulong>();
            }));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
        ↪  IList<ulong> sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
```

```
759                 if (sequence.Count > 0)
760                 {
761                     Links.EnsureLinkExists(sequence);
762                     var results = new HashSet<LinkIndex>();
763                     //var nextResults = new HashSet<ulong>();
764                     //for (var i = 0; i < sequence.Length; i++)
765                     //{
766                     //    AllUsagesCore(sequence[i], nextResults);
767                     //    if (results.IsNullOrEmpty())
768                     //    {
769                     //        results = nextResults;
770                     //        nextResults = new HashSet<ulong>();
771                     //    }
772                     //    else
773                     //    {
774                     //        results.IntersectWith(nextResults);
775                     //        nextResults.Clear();
776                     //    }
777                     //}
778                     var collector1 = new AllUsagesCollector1(Links.Unsync, results);
779                     collector1.Collect(Links.Unsync.GetLink(sequence[0]));
780                     var next = new HashSet<ulong>();
781                     for (var i = 1; i < sequence.Count; i++)
782                     {
783                         var collector = new AllUsagesCollector1(Links.Unsync, next);
784                         collector.Collect(Links.Unsync.GetLink(sequence[i]));
785 
786                         results.IntersectWith(next);
787                         next.Clear();
788                     }
789                     var filteredResults = new HashSet<ulong>();
790                     var matcher = new Matcher(this, sequence, filteredResults, null,
                        ↪  readAsElements);
791                     matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
                        ↪  x)); // OrderBy is a Hack
792                     return filteredResults;
793                 }
794                 return new HashSet<ulong>();
795             });
796         }
797 
798         // Does not work
799         //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
            ↪  params ulong[] sequence)
800         //{
801         //    var visited = new HashSet<ulong>();
802         //    var results = new HashSet<ulong>();
803         //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
            ↪  true; }, readAsElements);
804         //    var last = sequence.Length - 1;
805         //    for (var i = 0; i < last; i++)
806         //    {
807         //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
808         //    }
809         //    return results;
810         //}
811 
812         [MethodImpl(MethodImplOptions.AggressiveInlining)]
813         public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
814         {
815             return _sync.ExecuteReadOperation(() =>
816             {
817                 if (sequence.Length > 0)
818                 {
819                     Links.EnsureLinkExists(sequence);
820                     //var firstElement = sequence[0];
821                     //if (sequence.Length == 1)
822                     //{
823                     //    //results.Add(firstElement);
824                     //    return results;
825                     //}
826                     //if (sequence.Length == 2)
827                     //{
828                     //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
829                     //    //if (doublet != Doublets.Links.Null)
830                     //    //    results.Add(doublet);
831                     //    return results;
832                     //}
```

```
833                     //var lastElement = sequence[sequence.Length - 1];
834                     //Func<ulong, bool> handler = x =>
835                     //{
836                     //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
                        ↪  results.Add(x);
837                     //    return true;
838                     //};
839                     //if (sequence.Length >= 2)
840                     //    StepRight(handler, sequence[0], sequence[1]);
841                     //var last = sequence.Length - 2;
842                     //for (var i = 1; i < last; i++)
843                     //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
844                     //if (sequence.Length >= 3)
845                     //    StepLeft(handler, sequence[sequence.Length - 2],
                        ↪  sequence[sequence.Length - 1]);
846                     //////if (sequence.Length == 1)
847                     //////{
848                     //////    throw new NotImplementedException(); // all sequences, containing
                        ↪  this element?
849                     //////}
850                     //////if (sequence.Length == 2)
851                     //////{
852                     //////    var results = new List<ulong>();
853                     //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
854                     //////    return results;
855                     //////}
856                     //////var matches = new List<List<ulong>>();
857                     //////var last = sequence.Length - 1;
858                     //////for (var i = 0; i < last; i++)
859                     //////{
860                     //////    var results = new List<ulong>();
861                     //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
862                     //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
863                     //////    if (results.Count > 0)
864                     //////        matches.Add(results);
865                     //////    else
866                     //////        return results;
867                     //////    if (matches.Count == 2)
868                     //////    {
869                     //////        var merged = new List<ulong>();
870                     //////        for (var j = 0; j < matches[0].Count; j++)
871                     //////            for (var k = 0; k < matches[1].Count; k++)
872                     //////                CloseInnerConnections(merged.Add, matches[0][j],
                        ↪  matches[1][k]);
873                     //////        if (merged.Count > 0)
874                     //////            matches = new List<List<ulong>> { merged };
875                     //////        else
876                     //////            return new List<ulong>();
877                     //////    }
878                     //////}
879                     //////if (matches.Count > 0)
880                     //////{
881                     //////    var usages = new HashSet<ulong>();
882                     //////    for (int i = 0; i < sequence.Length; i++)
883                     //////    {
884                     //////        AllUsagesCore(sequence[i], usages);
885                     //////    }
886                     //////    //for (int i = 0; i < matches[0].Count; i++)
887                     //////    //    AllUsagesCore(matches[0][i], usages);
888                     //////    //usages.UnionWith(matches[0]);
889                     //////    return usages.ToList();
890                     //////}
891                     var firstLinkUsages = new HashSet<ulong>();
892                     AllUsagesCore(sequence[0], firstLinkUsages);
893                     firstLinkUsages.Add(sequence[0]);
894                     //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
                        ↪  sequence[0] }; // or all sequences, containing this element?
895                     //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
                        ↪  1).ToList();
896                     var results = new HashSet<ulong>();
897                     foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
                        ↪  firstLinkUsages, 1))
898                     {
899                         AllUsagesCore(match, results);
900                     }
901                     return results.ToList();
902                 }
```

```
903                 return new List<ulong>();
904             });
905         }
906 
907         /// <remarks>
908         /// TODO: Может потробоваться ограничение на уровень глубины рекурсии
909         /// </remarks>
910         [MethodImpl(MethodImplOptions.AggressiveInlining)]
911         public HashSet<ulong> AllUsages(ulong link)
912         {
913             return _sync.ExecuteReadOperation(() =>
914             {
915                 var usages = new HashSet<ulong>();
916                 AllUsagesCore(link, usages);
917                 return usages;
918             });
919         }
920 
921         // При сборе всех использований (последовательностей) можно сохранять обратный путь к
        ↪  той связи с которой начинался поиск (STTTSSSTT),
922         // причём достаточно одного бита для хранения перехода влево или вправо
923         [MethodImpl(MethodImplOptions.AggressiveInlining)]
924         private void AllUsagesCore(ulong link, HashSet<ulong> usages)
925         {
926             bool handler(ulong doublet)
927             {
928                 if (usages.Add(doublet))
929                 {
930                     AllUsagesCore(doublet, usages);
931                 }
932                 return true;
933             }
934             Links.Unsync.Each(link, Constants.Any, handler);
935             Links.Unsync.Each(Constants.Any, link, handler);
936         }
937 
938         [MethodImpl(MethodImplOptions.AggressiveInlining)]
939         public HashSet<ulong> AllBottomUsages(ulong link)
940         {
941             return _sync.ExecuteReadOperation(() =>
942             {
943                 var visits = new HashSet<ulong>();
944                 var usages = new HashSet<ulong>();
945                 AllBottomUsagesCore(link, visits, usages);
946                 return usages;
947             });
948         }
949 
950         [MethodImpl(MethodImplOptions.AggressiveInlining)]
951         private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
        ↪  usages)
952         {
953             bool handler(ulong doublet)
954             {
955                 if (visits.Add(doublet))
956                 {
957                     AllBottomUsagesCore(doublet, visits, usages);
958                 }
959                 return true;
960             }
961             if (Links.Unsync.Count(Constants.Any, link) == 0)
962             {
963                 usages.Add(link);
964             }
965             else
966             {
967                 Links.Unsync.Each(link, Constants.Any, handler);
968                 Links.Unsync.Each(Constants.Any, link, handler);
969             }
970         }
971 
972         [MethodImpl(MethodImplOptions.AggressiveInlining)]
973         public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
974         {
975             if (Options.UseSequenceMarker)
976             {
977                 var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                    ↪  Options.MarkedSequenceMatcher, symbol);
```

```
978                 return counter.Count();
979             }
980             else
981             {
982                 var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                    ↪  symbol);
983                 return counter.Count();
984             }
985         }
986 
987         [MethodImpl(MethodImplOptions.AggressiveInlining)]
988         private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
                ↪  LinkIndex> outerHandler)
989         {
990             bool handler(ulong doublet)
991             {
992                 if (usages.Add(doublet))
993                 {
994                     if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
995                     {
996                         return false;
997                     }
998                     if (!AllUsagesCore1(doublet, usages, outerHandler))
999                     {
1000                        return false;
1001                    }
1002                }
1003                return true;
1004            }
1005            return Links.Unsync.Each(link, Constants.Any, handler)
1006                && Links.Unsync.Each(Constants.Any, link, handler);
1007        }
1008
1009        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1010        public void CalculateAllUsages(ulong[] totals)
1011        {
1012            var calculator = new AllUsagesCalculator(Links, totals);
1013            calculator.Calculate();
1014        }
1015
1016        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1017        public void CalculateAllUsages2(ulong[] totals)
1018        {
1019            var calculator = new AllUsagesCalculator2(Links, totals);
1020            calculator.Calculate();
1021        }
1022
1023        private class AllUsagesCalculator
1024        {
1025            private readonly SynchronizedLinks<ulong> _links;
1026            private readonly ulong[] _totals;
1027
1028            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029            public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1030            {
1031                _links = links;
1032                _totals = totals;
1033            }
1034
1035            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1036            public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
                    ↪  CalculateCore);
1037
1038            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1039            private bool CalculateCore(ulong link)
1040            {
1041                if (_totals[link] == 0)
1042                {
1043                    var total = 1UL;
1044                    _totals[link] = total;
1045                    var visitedChildren = new HashSet<ulong>();
1046                    bool linkCalculator(ulong child)
1047                    {
1048                        if (link != child && visitedChildren.Add(child))
1049                        {
1050                            total += _totals[child] == 0 ? 1 : _totals[child];
1051                        }
1052                        return true;
1053                    }
```

```
                _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
                _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
                _totals[link] = total;
            }
            return true;
        }
    }

    private class AllUsagesCalculator2
    {
        private readonly SynchronizedLinks<ulong> _links;
        private readonly ulong[] _totals;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
        {
            _links = links;
            _totals = totals;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
        ↪  CalculateCore);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsElement(ulong link)
        {
            //_linksInSequence.Contains(link) ||
            return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
            ↪  link;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool CalculateCore(ulong link)
        {
            // TODO: Проработать защиту от зацикливания
            // Основано на SequenceWalker.WalkLeft
            Func<ulong, ulong> getSource = _links.Unsync.GetSource;
            Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
            Func<ulong, bool> isElement = IsElement;
            void visitLeaf(ulong parent)
            {
                if (link != parent)
                {
                    _totals[parent]++;
                }
            }
            void visitNode(ulong parent)
            {
                if (link != parent)
                {
                    _totals[parent]++;
                }
            }
            var stack = new Stack();
            var element = link;
            if (isElement(element))
            {
                visitLeaf(element);
            }
            else
            {
                while (true)
                {
                    if (isElement(element))
                    {
                        if (stack.Count == 0)
                        {
                            break;
                        }
                        element = stack.Pop();
                        var source = getSource(element);
                        var target = getTarget(element);
                        // Обработка элемента
                        if (isElement(target))
                        {
                            visitLeaf(target);
                        }
                        if (isElement(source))
```

```
                        {
                            visitLeaf(source);
                        }
                        element = source;
                    }
                    else
                    {
                        stack.Push(element);
                        visitNode(element);
                        element = getTarget(element);
                    }
                }
            }
            _totals[link]++;
            return true;
        }
    }

    private class AllUsagesCollector
    {
        private readonly ILinks<ulong> _links;
        private readonly HashSet<ulong> _usages;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
        {
            _links = links;
            _usages = usages;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Collect(ulong link)
        {
            if (_usages.Add(link))
            {
                _links.Each(link, _links.Constants.Any, Collect);
                _links.Each(_links.Constants.Any, link, Collect);
            }
            return true;
        }
    }

    private class AllUsagesCollector1
    {
        private readonly ILinks<ulong> _links;
        private readonly HashSet<ulong> _usages;
        private readonly ulong _continue;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
        {
            _links = links;
            _usages = usages;
            _continue = _links.Constants.Continue;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ulong Collect(IList<ulong> link)
        {
            var linkIndex = _links.GetIndex(link);
            if (_usages.Add(linkIndex))
            {
                _links.Each(Collect, _links.Constants.Any, linkIndex);
            }
            return _continue;
        }
    }

    private class AllUsagesCollector2
    {
        private readonly ILinks<ulong> _links;
        private readonly BitString _usages;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
        {
            _links = links;
            _usages = usages;
        }

```

```
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Collect(ulong link)
            {
                if (_usages.Add((long)link))
                {
                    _links.Each(link, _links.Constants.Any, Collect);
                    _links.Each(_links.Constants.Any, link, Collect);
                }
                return true;
            }
        }

        private class AllUsagesIntersectingCollector
        {
            private readonly SynchronizedLinks<ulong> _links;
            private readonly HashSet<ulong> _intersectWith;
            private readonly HashSet<ulong> _usages;
            private readonly HashSet<ulong> _enter;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
            ↪  intersectWith, HashSet<ulong> usages)
            {
                _links = links;
                _intersectWith = intersectWith;
                _usages = usages;
                _enter = new HashSet<ulong>(); // защита от зацикливания
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Collect(ulong link)
            {
                if (_enter.Add(link))
                {
                    if (_intersectWith.Contains(link))
                    {
                        _usages.Add(link);
                    }
                    _links.Unsync.Each(link, _links.Constants.Any, Collect);
                    _links.Unsync.Each(_links.Constants.Any, link, Collect);
                }
                return true;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
        ↪  right)
        {
            TryStepLeftUp(handler, left, right);
            TryStepRightUp(handler, right, left);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
        ↪  right)
        {
            // Direct
            if (left == right)
            {
                handler(new LinkAddress<LinkIndex>(left));
            }
            var doublet = Links.Unsync.SearchOrDefault(left, right);
            if (doublet != Constants.Null)
            {
                handler(new LinkAddress<LinkIndex>(doublet));
            }
            // Inner
            CloseInnerConnections(handler, left, right);
            // Outer
            StepLeft(handler, left, right);
            StepRight(handler, left, right);
            PartialStepRight(handler, left, right);
            PartialStepLeft(handler, left, right);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
        ↪  HashSet<ulong> previousMatchings, long startAt)
```

```
        {
            if (startAt >= sequence.Length) // ?
            {
                return previousMatchings;
            }
            var secondLinkUsages = new HashSet<ulong>();
            AllUsagesCore(sequence[startAt], secondLinkUsages);
            secondLinkUsages.Add(sequence[startAt]);
            var matchings = new HashSet<ulong>();
            var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
            //for (var i = 0; i < previousMatchings.Count; i++)
            foreach (var secondLinkUsage in secondLinkUsages)
            {
                foreach (var previousMatching in previousMatchings)
                {
                    //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
                    ↪  secondLinkUsage);
                    StepRight(filler.AddFirstAndReturnConstant, previousMatching,
                    ↪  secondLinkUsage);
                    TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
                    ↪  previousMatching);
                    //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
                    ↪  sequence[startAt]); // почему-то эта ошибочная запись приводит к
                    ↪  желаемым результам.
                    PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
                    ↪  secondLinkUsage);
                }
            }
            if (matchings.Count == 0)
            {
                return matchings;
            }
            return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
        ↪  links, params ulong[] sequence)
        {
            if (sequence == null)
            {
                return;
            }
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
                ↪  !links.Exists(sequence[i]))
                {
                    throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                    ↪  $"patternSequence[{i}]");
                }
            }
        }

        // Pattern Matching -> Key To Triggers
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                patternSequence = Simplify(patternSequence);
                if (patternSequence.Length > 0)
                {
                    EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
                    var uniqueSequenceElements = new HashSet<ulong>();
                    for (var i = 0; i < patternSequence.Length; i++)
                    {
                        if (patternSequence[i] != Constants.Any && patternSequence[i] !=
                        ↪  ZeroOrMany)
                        {
                            uniqueSequenceElements.Add(patternSequence[i]);
                        }
                    }
                    var results = new HashSet<ulong>();
                    foreach (var uniqueSequenceElement in uniqueSequenceElements)
                    {
                        AllUsagesCore(uniqueSequenceElement, results);
```

```
                }
                var filteredResults = new HashSet<ulong>();
                var matcher = new PatternMatcher(this, patternSequence, filteredResults);
                matcher.AddAllPatternMatchedToResults(results);
                return filteredResults;
            }
            return new HashSet<ulong>();
        });
    }

    // Найти все возможные связи между указанным списком связей.
    // Находит связи между всеми указанными связями в любом порядке.
    // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
    ↪  несколько раз в последовательности)
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            var results = new HashSet<ulong>();
            if (linksToConnect.Length > 0)
            {
                Links.EnsureLinkExists(linksToConnect);
                AllUsagesCore(linksToConnect[0], results);
                for (var i = 1; i < linksToConnect.Length; i++)
                {
                    var next = new HashSet<ulong>();
                    AllUsagesCore(linksToConnect[i], next);
                    results.IntersectWith(next);
                }
            }
            return results;
        });
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            var results = new HashSet<ulong>();
            if (linksToConnect.Length > 0)
            {
                Links.EnsureLinkExists(linksToConnect);
                var collector1 = new AllUsagesCollector(Links.Unsync, results);
                collector1.Collect(linksToConnect[0]);
                var next = new HashSet<ulong>();
                for (var i = 1; i < linksToConnect.Length; i++)
                {
                    var collector = new AllUsagesCollector(Links.Unsync, next);
                    collector.Collect(linksToConnect[i]);
                    results.IntersectWith(next);
                    next.Clear();
                }
            }
            return results;
        });
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            var results = new HashSet<ulong>();
            if (linksToConnect.Length > 0)
            {
                Links.EnsureLinkExists(linksToConnect);
                var collector1 = new AllUsagesCollector(Links, results);
                collector1.Collect(linksToConnect[0]);
                //AllUsagesCore(linksToConnect[0], results);
                for (var i = 1; i < linksToConnect.Length; i++)
                {
                    var next = new HashSet<ulong>();
                    var collector = new AllUsagesIntersectingCollector(Links, results, next);
                    collector.Collect(linksToConnect[i]);
                    //AllUsagesCore(linksToConnect[i], next);
                    //results.IntersectWith(next);
```

```
                    results = next;
                }
            }
            return results;
        });
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            var results = new BitString((long)Links.Unsync.Count() + 1); // new
            ↪  BitArray((int)_links.Total + 1);
            if (linksToConnect.Length > 0)
            {
                Links.EnsureLinkExists(linksToConnect);
                var collector1 = new AllUsagesCollector2(Links.Unsync, results);
                collector1.Collect(linksToConnect[0]);
                for (var i = 1; i < linksToConnect.Length; i++)
                {
                    var next = new BitString((long)Links.Unsync.Count() + 1); //new
                    ↪  BitArray((int)_links.Total + 1);
                    var collector = new AllUsagesCollector2(Links.Unsync, next);
                    collector.Collect(linksToConnect[i]);
                    results = results.And(next);
                }
            }
            return results.GetSetUInt64Indices();
        });
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private static ulong[] Simplify(ulong[] sequence)
    {
        // Считаем новый размер последовательности
        long newLength = 0;
        var zeroOrManyStepped = false;
        for (var i = 0; i < sequence.Length; i++)
        {
            if (sequence[i] == ZeroOrMany)
            {
                if (zeroOrManyStepped)
                {
                    continue;
                }
                zeroOrManyStepped = true;
            }
            else
            {
                //if (zeroOrManyStepped) Is it efficient?
                zeroOrManyStepped = false;
            }
            newLength++;
        }
        // Строим новую последовательность
        zeroOrManyStepped = false;
        var newSequence = new ulong[newLength];
        long j = 0;
        for (var i = 0; i < sequence.Length; i++)
        {
            //var current = zeroOrManyStepped;
            //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
            //if (current && zeroOrManyStepped)
            //    continue;
            //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
            //if (zeroOrManyStepped && newZeroOrManyStepped)
            //    continue;
            //zeroOrManyStepped = newZeroOrManyStepped;
            if (sequence[i] == ZeroOrMany)
            {
                if (zeroOrManyStepped)
                {
                    continue;
                }
                zeroOrManyStepped = true;
            }
            else
            {
```

```
                    //if (zeroOrManyStepped) Is it efficient?
                    zeroOrManyStepped = false;
                }
                newSequence[j++] = sequence[i];
            }
            return newSequence;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void TestSimplify()
        {
            var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
            ↪  ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
            var simplifiedSequence = Simplify(sequence);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetSimilarSequences() => new List<ulong>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Prediction()
        {
            //_links
            //sequences
        }

        #region From Triplets

        //public static void DeleteSequence(Link sequence)
        //{
        //}

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> CollectMatchingSequences(ulong[] links)
        {
            if (links.Length == 1)
            {
                throw new InvalidOperationException("Подпоследовательности с одним элементом не
                ↪  поддерживаются.");
            }
            var leftBound = 0;
            var rightBound = links.Length - 1;
            var left = links[leftBound++];
            var right = links[rightBound--];
            var results = new List<ulong>();
            CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
            return results;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
        ↪  middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
        {
            var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
            var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
            if (leftLinkTotalReferers <= rightLinkTotalReferers)
            {
                var nextLeftLink = middleLinks[leftBound];
                var elements = GetRightElements(leftLink, nextLeftLink);
                if (leftBound <= rightBound)
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != 0)
                        {
                            CollectMatchingSequences(element, leftBound + 1, middleLinks,
                            ↪  rightLink, rightBound, ref results);
                        }
                    }
                }
                else
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != 0)
                        {
```

```
                        results.Add(element);
                    }
                }
            }
        }
        else
        {
            var nextRightLink = middleLinks[rightBound];
            var elements = GetLeftElements(rightLink, nextRightLink);
            if (leftBound <= rightBound)
            {
                for (var i = elements.Length - 1; i >= 0; i--)
                {
                    var element = elements[i];
                    if (element != 0)
                    {
                        CollectMatchingSequences(leftLink, leftBound, middleLinks,
                        ↪  elements[i], rightBound - 1, ref results);
                    }
                }
            }
            else
            {
                for (var i = elements.Length - 1; i >= 0; i--)
                {
                    var element = elements[i];
                    if (element != 0)
                    {
                        results.Add(element);
                    }
                }
            }
        }
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public ulong[] GetRightElements(ulong startLink, ulong rightLink)
    {
        var result = new ulong[5];
        TryStepRight(startLink, rightLink, result, 0);
        Links.Each(Constants.Any, startLink, couple =>
        {
            if (couple != startLink)
            {
                if (TryStepRight(couple, rightLink, result, 2))
                {
                    return false;
                }
            }
            return true;
        });
        if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
        {
            result[4] = startLink;
        }
        return result;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
    {
        var added = 0;
        Links.Each(startLink, Constants.Any, couple =>
        {
            if (couple != startLink)
            {
                var coupleTarget = Links.GetTarget(couple);
                if (coupleTarget == rightLink)
                {
                    result[offset] = couple;
                    if (++added == 2)
                    {
                        return false;
                    }
                }
                else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
                ↪  == Net.And &&
                {
```

```
                    result[offset + 1] = couple;
                    if (++added == 2)
                    {
                        return false;
                    }
                }
            }
            return true;
        });
        return added > 0;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
    {
        var result = new ulong[5];
        TryStepLeft(startLink, leftLink, result, 0);
        Links.Each(startLink, Constants.Any, couple =>
        {
            if (couple != startLink)
            {
                if (TryStepLeft(couple, leftLink, result, 2))
                {
                    return false;
                }
            }
            return true;
        });
        if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
        {
            result[4] = leftLink;
        }
        return result;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
    {
        var added = 0;
        Links.Each(Constants.Any, startLink, couple =>
        {
            if (couple != startLink)
            {
                var coupleSource = Links.GetSource(couple);
                if (coupleSource == leftLink)
                {
                    result[offset] = couple;
                    if (++added == 2)
                    {
                        return false;
                    }
                }
                else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
                ↪  == Net.And &&
                {
                    result[offset + 1] = couple;
                    if (++added == 2)
                    {
                        return false;
                    }
                }
            }
            return true;
        });
        return added > 0;
    }

    #endregion

    #region Walkers

    public class PatternMatcher : RightSequenceWalker<ulong>
    {
        private readonly Sequences _sequences;
        private readonly ulong[] _patternSequence;
        private readonly HashSet<LinkIndex> _linksInSequence;
        private readonly HashSet<LinkIndex> _results;

        #region Pattern Match

```

```
1736            enum PatternBlockType
1737            {
1738                Undefined,
1739                Gap,
1740                Elements
1741            }
1742
1743            struct PatternBlock
1744            {
1745                public PatternBlockType Type;
1746                public long Start;
1747                public long Stop;
1748            }
1749
1750            private readonly List<PatternBlock> _pattern;
1751            private int _patternPosition;
1752            private long _sequencePosition;
1753
1754            #endregion
1755
1756            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1757            public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
                ↪  HashSet<LinkIndex> results)
1758                : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1759            {
1760                _sequences = sequences;
1761                _patternSequence = patternSequence;
1762                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪  _sequences.Constants.Any && x != ZeroOrMany));
1763                _results = results;
1764                _pattern = CreateDetailedPattern();
1765            }
1766
1767            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1768            protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
                ↪  base.IsElement(link);
1769
1770            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1771            public bool PatternMatch(LinkIndex sequenceToMatch)
1772            {
1773                _patternPosition = 0;
1774                _sequencePosition = 0;
1775                foreach (var part in Walk(sequenceToMatch))
1776                {
1777                    if (!PatternMatchCore(part))
1778                    {
1779                        break;
1780                    }
1781                }
1782                return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
                    ↪  - 1 && _pattern[_patternPosition].Start == 0);
1783            }
1784
1785            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1786            private List<PatternBlock> CreateDetailedPattern()
1787            {
1788                var pattern = new List<PatternBlock>();
1789                var patternBlock = new PatternBlock();
1790                for (var i = 0; i < _patternSequence.Length; i++)
1791                {
1792                    if (patternBlock.Type == PatternBlockType.Undefined)
1793                    {
1794                        if (_patternSequence[i] == _sequences.Constants.Any)
1795                        {
1796                            patternBlock.Type = PatternBlockType.Gap;
1797                            patternBlock.Start = 1;
1798                            patternBlock.Stop = 1;
1799                        }
1800                        else if (_patternSequence[i] == ZeroOrMany)
1801                        {
1802                            patternBlock.Type = PatternBlockType.Gap;
1803                            patternBlock.Start = 0;
1804                            patternBlock.Stop = long.MaxValue;
1805                        }
1806                        else
1807                        {
1808                            patternBlock.Type = PatternBlockType.Elements;
1809                            patternBlock.Start = i;
1810                            patternBlock.Stop = i;
1811                        }
```

```
                }
                else if (patternBlock.Type == PatternBlockType.Elements)
                {
                    if (_patternSequence[i] == _sequences.Constants.Any)
                    {
                        pattern.Add(patternBlock);
                        patternBlock = new PatternBlock
                        {
                            Type = PatternBlockType.Gap,
                            Start = 1,
                            Stop = 1
                        };
                    }
                    else if (_patternSequence[i] == ZeroOrMany)
                    {
                        pattern.Add(patternBlock);
                        patternBlock = new PatternBlock
                        {
                            Type = PatternBlockType.Gap,
                            Start = 0,
                            Stop = long.MaxValue
                        };
                    }
                    else
                    {
                        patternBlock.Stop = i;
                    }
                }
                else // patternBlock.Type == PatternBlockType.Gap
                {
                    if (_patternSequence[i] == _sequences.Constants.Any)
                    {
                        patternBlock.Start++;
                        if (patternBlock.Stop < patternBlock.Start)
                        {
                            patternBlock.Stop = patternBlock.Start;
                        }
                    }
                    else if (_patternSequence[i] == ZeroOrMany)
                    {
                        patternBlock.Stop = long.MaxValue;
                    }
                    else
                    {
                        pattern.Add(patternBlock);
                        patternBlock = new PatternBlock
                        {
                            Type = PatternBlockType.Elements,
                            Start = i,
                            Stop = i
                        };
                    }
                }
            }
            if (patternBlock.Type != PatternBlockType.Undefined)
            {
                pattern.Add(patternBlock);
            }
            return pattern;
        }

        // match: search for regexp anywhere in text
        //int match(char* regexp, char* text)
        //{
        //    do
        //    {
        //    } while (*text++ != '\0');
        //    return 0;
        //}

        // matchhere: search for regexp at beginning of text
        //int matchhere(char* regexp, char* text)
        //{
        //    if (regexp[0] == '\0')
        //        return 1;
        //    if (regexp[1] == '*')
        //        return matchstar(regexp[0], regexp + 2, text);
        //    if (regexp[0] == '$' && regexp[1] == '\0')
        //        return *text == '\0';
```

```
//    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
//        return matchhere(regexp + 1, text + 1);
//    return 0;
//}

// matchstar: search for c*regexp at beginning of text
//int matchstar(int c, char* regexp, char* text)
//{
//    do
//    {    /* a * matches zero or more instances */
//        if (matchhere(regexp, text))
//            return 1;
//    } while (*text != '\0' && (*text++ == c || c == '.'));
//    return 0;
//}

//private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out long maximumGap)
//{
//    mininumGap = 0;
//    maximumGap = 0;
//    element = 0;
//    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
//    {
//        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
//            mininumGap++;
//        else if (_patternSequence[_patternPosition] == ZeroOrMany)
//            maximumGap = long.MaxValue;
//        else
//            break;
//    }

//    if (maximumGap < mininumGap)
//        maximumGap = mininumGap;
//}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
private bool PatternMatchCore(LinkIndex element)
{
    if (_patternPosition >= _pattern.Count)
    {
        _patternPosition = -2;
        return false;
    }
    var currentPatternBlock = _pattern[_patternPosition];
    if (currentPatternBlock.Type == PatternBlockType.Gap)
    {
        //var currentMatchingBlockLength = (_sequencePosition - _lastMatchedBlockPosition);
        if (_sequencePosition < currentPatternBlock.Start)
        {
            _sequencePosition++;
            return true; // Двигаемся дальше
        }
        // Это последний блок
        if (_pattern.Count == _patternPosition + 1)
        {
            _patternPosition++;
            _sequencePosition = 0;
            return false; // Полное соответствие
        }
        else
        {
            if (_sequencePosition > currentPatternBlock.Stop)
            {
                return false; // Соответствие невозможно
            }
            var nextPatternBlock = _pattern[_patternPosition + 1];
            if (_patternSequence[nextPatternBlock.Start] == element)
            {
                if (nextPatternBlock.Start < nextPatternBlock.Stop)
                {
                    _patternPosition++;
                    _sequencePosition = 1;
                }
                else
                {
                    _patternPosition += 2;
                    _sequencePosition = 0;
```

```
                            }
                        }
                    }
                }
                else // currentPatternBlock.Type == PatternBlockType.Elements
                {
                    var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
                    if (_patternSequence[patternElementPosition] != element)
                    {
                        return false; // Соответствие невозможно
                    }
                    if (patternElementPosition == currentPatternBlock.Stop)
                    {
                        _patternPosition++;
                        _sequencePosition = 0;
                    }
                    else
                    {
                        _sequencePosition++;
                    }
                }
                return true;
                //if (_patternSequence[_patternPosition] != element)
                //    return false;
                //else
                //{
                //    _sequencePosition++;
                //    _patternPosition++;
                //    return true;
                //}
                ////////
                //if (_filterPosition == _patternSequence.Length)
                //{
                //    _filterPosition = -2; // Длиннее чем нужно
                //    return false;
                //}
                //if (element != _patternSequence[_filterPosition])
                //{
                //    _filterPosition = -1;
                //    return false; // Начинается иначе
                //}
                //_filterPosition++;
                //if (_filterPosition == (_patternSequence.Length - 1))
                //    return false;
                //if (_filterPosition >= 0)
                //{
                //    if (element == _patternSequence[_filterPosition + 1])
                //        _filterPosition++;
                //    else
                //        return false;
                //}
                //if (_filterPosition < 0)
                //{
                //    if (element == _patternSequence[0])
                //        _filterPosition = 0;
                //}
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
            {
                foreach (var sequenceToMatch in sequencesToMatch)
                {
                    if (PatternMatch(sequenceToMatch))
                    {
                        _results.Add(sequenceToMatch);
                    }
                }
            }
        }

        #endregion
    }
}
```

1.149 ./csharp/Platform.Data.Doublets/Sequences/Sequences.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
```

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

```
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
76          public bool IsSequence(LinkIndex sequence)
77          {
78              return _sync.ExecuteReadOperation(() =>
79              {
80                  if (Options.UseSequenceMarker)
81                  {
82                      return Options.MarkedSequenceMatcher.IsMatched(sequence);
83                  }
84                  return !Links.Unsync.IsPartialPoint(sequence);
85              });
86          }
87
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          private LinkIndex GetSequenceByElements(LinkIndex sequence)
90          {
91              if (Options.UseSequenceMarker)
92              {
93                  return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
94              }
95              return sequence;
96          }
97
98          [MethodImpl(MethodImplOptions.AggressiveInlining)]
99          private LinkIndex GetSequenceElements(LinkIndex sequence)
100         {
101             if (Options.UseSequenceMarker)
102             {
103                 var linkContents = new Link<ulong>(Links.GetLink(sequence));
104                 if (linkContents.Source == Options.SequenceMarkerLink)
105                 {
106                     return linkContents.Target;
107                 }
108                 if (linkContents.Target == Options.SequenceMarkerLink)
109                 {
110                     return linkContents.Source;
111                 }
112             }
113             return sequence;
114         }
115
116         #region Count
117
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
119         public LinkIndex Count(IList<LinkIndex> restrictions)
120         {
121             if (restrictions.IsNullOrEmpty())
122             {
123                 return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
124             }
125             if (restrictions.Count == 1) // Первая связь это адрес
126             {
127                 var sequenceIndex = restrictions[0];
128                 if (sequenceIndex == Constants.Null)
129                 {
130                     return 0;
131                 }
132                 if (sequenceIndex == Constants.Any)
133                 {
134                     return Count(null);
135                 }
136                 if (Options.UseSequenceMarker)
137                 {
138                     return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
139                 }
140                 return Links.Exists(sequenceIndex) ? 1UL : 0;
141             }
142             throw new NotImplementedException();
143         }
144
145         [MethodImpl(MethodImplOptions.AggressiveInlining)]
146         private LinkIndex CountUsages(params LinkIndex[] restrictions)
147         {
148             if (restrictions.Length == 0)
149             {
150                 return 0;
151             }
152             if (restrictions.Length == 1) // Первая связь это адрес
153             {
```

```
154                 if (restrictions[0] == Constants.Null)
155                 {
156                     return 0;
157                 }
158                 var any = Constants.Any;
159                 if (Options.UseSequenceMarker)
160                 {
161                     var elementsLink = GetSequenceElements(restrictions[0]);
162                     var sequenceLink = GetSequenceByElements(elementsLink);
163                     if (sequenceLink != Constants.Null)
164                     {
165                         return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
                            ↪ 1;
166                     }
167                     return Links.Count(any, elementsLink);
168                 }
169                 return Links.Count(any, restrictions[0]);
170             }
171             throw new NotImplementedException();
172         }
173 
174         #endregion
175 
176         #region Create
177 
178         [MethodImpl(MethodImplOptions.AggressiveInlining)]
179         public LinkIndex Create(IList<LinkIndex> restrictions)
180         {
181             return _sync.ExecuteWriteOperation(() =>
182             {
183                 if (restrictions.IsNullOrEmpty())
184                 {
185                     return Constants.Null;
186                 }
187                 Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
188                 return CreateCore(restrictions);
189             });
190         }
191 
192         [MethodImpl(MethodImplOptions.AggressiveInlining)]
193         private LinkIndex CreateCore(IList<LinkIndex> restrictions)
194         {
195             LinkIndex[] sequence = restrictions.SkipFirst();
196             if (Options.UseIndex)
197             {
198                 Options.Index.Add(sequence);
199             }
200             var sequenceRoot = default(LinkIndex);
201             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
202             {
203                 var matches = Each(restrictions);
204                 if (matches.Count > 0)
205                 {
206                     sequenceRoot = matches[0];
207                 }
208             }
209             else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
210             {
211                 return CompactCore(sequence);
212             }
213             if (sequenceRoot == default)
214             {
215                 sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
216             }
217             if (Options.UseSequenceMarker)
218             {
219                 return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
220             }
221             return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
222         }
223 
224         #endregion
225 
226         #region Each
227 
228         [MethodImpl(MethodImplOptions.AggressiveInlining)]
229         public List<LinkIndex> Each(IList<LinkIndex> sequence)
230         {
231             var results = new List<LinkIndex>();
```

```
232             var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
233             Each(filler.AddFirstAndReturnConstant, sequence);
234             return results;
235         }
236 
237         [MethodImpl(MethodImplOptions.AggressiveInlining)]
238         public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
            ↪ restrictions)
239         {
240             return _sync.ExecuteReadOperation(() =>
241             {
242                 if (restrictions.IsNullOrEmpty())
243                 {
244                     return Constants.Continue;
245                 }
246                 Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
247                 if (restrictions.Count == 1)
248                 {
249                     var link = restrictions[0];
250                     var any = Constants.Any;
251                     if (link == any)
252                     {
253                         if (Options.UseSequenceMarker)
254                         {
255                             return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
                                ↪ Options.SequenceMarkerLink, any));
256                         }
257                         else
258                         {
259                             return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
                                ↪ any));
260                         }
261                     }
262                     if (Options.UseSequenceMarker)
263                     {
264                         var sequenceLinkValues = Links.Unsync.GetLink(link);
265                         if (sequenceLinkValues[Constants.SourcePart] ==
                            ↪ Options.SequenceMarkerLink)
266                         {
267                             link = sequenceLinkValues[Constants.TargetPart];
268                         }
269                     }
270                     var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
271                     sequence[0] = link;
272                     return handler(sequence);
273                 }
274                 else if (restrictions.Count == 2)
275                 {
276                     throw new NotImplementedException();
277                 }
278                 else if (restrictions.Count == 3)
279                 {
280                     return Links.Unsync.Each(handler, restrictions);
281                 }
282                 else
283                 {
284                     var sequence = restrictions.SkipFirst();
285                     if (Options.UseIndex && !Options.Index.MightContain(sequence))
286                     {
287                         return Constants.Break;
288                     }
289                     return EachCore(handler, sequence);
290                 }
291             });
292         }
293 
294         [MethodImpl(MethodImplOptions.AggressiveInlining)]
295         private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
            ↪ values)
296         {
297             var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
298             // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
                ↪ Id.
299             Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
                ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
                ↪ matcher.HandleFullMatched;
300             //if (sequence.Length >= 2)
301             if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
```

```
302             {
303                 return Constants.Break;
304             }
305             var last = values.Count - 2;
306             for (var i = 1; i < last; i++)
307             {
308                 if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
                    ↪  Constants.Continue)
309                 {
310                     return Constants.Break;
311                 }
312             }
313             if (values.Count >= 3)
314             {
315                 if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
                    ↪  != Constants.Continue)
316                 {
317                     return Constants.Break;
318                 }
319             }
320             return Constants.Continue;
321         }
322
323         [MethodImpl(MethodImplOptions.AggressiveInlining)]
324         private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪  left, LinkIndex right)
325         {
326             return Links.Unsync.Each(doublet =>
327             {
328                 var doubletIndex = doublet[Constants.IndexPart];
329                 if (StepRight(handler, doubletIndex, right) != Constants.Continue)
330                 {
331                     return Constants.Break;
332                 }
333                 if (left != doubletIndex)
334                 {
335                     return PartialStepRight(handler, doubletIndex, right);
336                 }
337                 return Constants.Continue;
338             }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
339         }
340
341         [MethodImpl(MethodImplOptions.AggressiveInlining)]
342         private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
            ↪  LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
            ↪  rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
            ↪  Constants.Any));
343
344         [MethodImpl(MethodImplOptions.AggressiveInlining)]
345         private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪  right, LinkIndex stepFrom)
346         {
347             var upStep = stepFrom;
348             var firstSource = Links.Unsync.GetTarget(upStep);
349             while (firstSource != right && firstSource != upStep)
350             {
351                 upStep = firstSource;
352                 firstSource = Links.Unsync.GetSource(upStep);
353             }
354             if (firstSource == right)
355             {
356                 return handler(new LinkAddress<LinkIndex>(stepFrom));
357             }
358             return Constants.Continue;
359         }
360
361         [MethodImpl(MethodImplOptions.AggressiveInlining)]
362         private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
            ↪  LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
            ↪  leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
            ↪  right));
363
364         [MethodImpl(MethodImplOptions.AggressiveInlining)]
365         private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪  left, LinkIndex stepFrom)
366         {
367             var upStep = stepFrom;
368             var firstTarget = Links.Unsync.GetSource(upStep);
```

```
            while (firstTarget != left && firstTarget != upStep)
            {
                upStep = firstTarget;
                firstTarget = Links.Unsync.GetTarget(upStep);
            }
            if (firstTarget == left)
            {
                return handler(new LinkAddress<LinkIndex>(stepFrom));
            }
            return Constants.Continue;
        }

        #endregion

        #region Update

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
        {
            var sequence = restrictions.SkipFirst();
            var newSequence = substitution.SkipFirst();
            if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
            {
                return Constants.Null;
            }
            if (sequence.IsNullOrEmpty())
            {
                return Create(substitution);
            }
            if (newSequence.IsNullOrEmpty())
            {
                Delete(restrictions);
                return Constants.Null;
            }
            return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
            {
                ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
                Links.EnsureLinkExists(newSequence);
                return UpdateCore(sequence, newSequence);
            }));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
        {
            LinkIndex bestVariant;
            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
            ↪  !sequence.EqualTo(newSequence))
            {
                bestVariant = CompactCore(newSequence);
            }
            else
            {
                bestVariant = CreateCore(newSequence);
            }
            // TODO: Check all options only ones before loop execution
            // Возможно нужно две версии Each, возвращающий фактические последовательности и с
            ↪  маркером,
            // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
            ↪  можно получить имея только фактические последовательности.
            foreach (var variant in Each(sequence))
            {
                if (variant != bestVariant)
                {
                    UpdateOneCore(variant, bestVariant);
                }
            }
            return bestVariant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(sequence);
                var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                var newSequenceElements = GetSequenceElements(newSequence);
```

```
                var newSequenceLink = GetSequenceByElements(newSequenceElements);
                if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                {
                    if (sequenceLink != Constants.Null)
                    {
                        Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
                    }
                    Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(sequence);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    var newSequenceElements = GetSequenceElements(newSequence);
                    var newSequenceLink = GetSequenceByElements(newSequenceElements);
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        if (sequenceLink != Constants.Null)
                        {
                            Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
                        }
                        Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
                    }
                }
                else
                {
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        Links.Unsync.MergeAndDelete(sequence, newSequence);
                    }
                }
            }
        }

        #endregion

        #region Delete

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Delete(IList<LinkIndex> restrictions)
        {
            _sync.ExecuteWriteOperation(() =>
            {
                var sequence = restrictions.SkipFirst();
                // TODO: Check all options only ones before loop execution
                foreach (var linkToDelete in Each(sequence))
                {
                    DeleteOneCore(linkToDelete);
                }
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void DeleteOneCore(LinkIndex link)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(link);
                var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                if (Options.UseCascadeDelete || CountUsages(link) == 0)
                {
                    if (sequenceLink != Constants.Null)
                    {
                        Links.Unsync.Delete(sequenceLink);
                    }
                    Links.Unsync.Delete(link);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
```

```
523                 if (Options.UseSequenceMarker)
524                 {
525                     var sequenceElements = GetSequenceElements(link);
526                     var sequenceLink = GetSequenceByElements(sequenceElements);
527                     if (Options.UseCascadeDelete || CountUsages(link) == 0)
528                     {
529                         if (sequenceLink != Constants.Null)
530                         {
531                             Links.Unsync.Delete(sequenceLink);
532                         }
533                         Links.Unsync.Delete(link);
534                     }
535                 }
536                 else
537                 {
538                     if (Options.UseCascadeDelete || CountUsages(link) == 0)
539                     {
540                         Links.Unsync.Delete(link);
541                     }
542                 }
543             }
544         }
545 
546         #endregion
547 
548         #region Compactification
549 
550         [MethodImpl(MethodImplOptions.AggressiveInlining)]
551         public void CompactAll()
552         {
553             _sync.ExecuteWriteOperation(() =>
554             {
555                 var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
556                 for (int i = 0; i < sequences.Count; i++)
557                 {
558                     var sequence = this.ToList(sequences[i]);
559                     Compact(sequence.ShiftRight());
560                 }
561             });
562         }
563 
564         /// <remarks>
565         /// bestVariant можно выбирать по максимальному числу использований,
566         /// но балансированный позволяет гарантировать уникальность (если есть возможность,
567         /// гарантировать его использование в других местах).
568         ///
569         /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
570         /// </remarks>
571         [MethodImpl(MethodImplOptions.AggressiveInlining)]
572         public LinkIndex Compact(IList<LinkIndex> sequence)
573         {
574             return _sync.ExecuteWriteOperation(() =>
575             {
576                 if (sequence.IsNullOrEmpty())
577                 {
578                     return Constants.Null;
579                 }
580                 Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
581                 return CompactCore(sequence);
582             });
583         }
584 
585         [MethodImpl(MethodImplOptions.AggressiveInlining)]
586         private LinkIndex CompactCore(IList<LinkIndex> sequence) => UpdateCore(sequence,
          ↪  sequence);
587 
588         #endregion
589 
590         #region Garbage Collection
591 
592         /// <remarks>
593         /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
          ↪  определить извне или в унаследованном классе
594         /// </remarks>
595         [MethodImpl(MethodImplOptions.AggressiveInlining)]
596         private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
          ↪  !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
597 
598         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
599         private void ClearGarbage(LinkIndex link)
600         {
601             if (IsGarbage(link))
602             {
603                 var contents = new Link<ulong>(Links.GetLink(link));
604                 Links.Unsync.Delete(link);
605                 ClearGarbage(contents.Source);
606                 ClearGarbage(contents.Target);
607             }
608         }
609 
610         #endregion
611 
612         #region Walkers
613 
614         [MethodImpl(MethodImplOptions.AggressiveInlining)]
615         public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
616         {
617             return _sync.ExecuteReadOperation(() =>
618             {
619                 var links = Links.Unsync;
620                 foreach (var part in Options.Walker.Walk(sequence))
621                 {
622                     if (!handler(part))
623                     {
624                         return false;
625                     }
626                 }
627                 return true;
628             });
629         }
630 
631         public class Matcher : RightSequenceWalker<LinkIndex>
632         {
633             private readonly Sequences _sequences;
634             private readonly IList<LinkIndex> _patternSequence;
635             private readonly HashSet<LinkIndex> _linksInSequence;
636             private readonly HashSet<LinkIndex> _results;
637             private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
638             private readonly HashSet<LinkIndex> _readAsElements;
639             private int _filterPosition;
640 
641             [MethodImpl(MethodImplOptions.AggressiveInlining)]
642             public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
                ↪  HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
                ↪  HashSet<LinkIndex> readAsElements = null)
643                 : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
644             {
645                 _sequences = sequences;
646                 _patternSequence = patternSequence;
647                 _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪  _links.Constants.Any && x != ZeroOrMany));
648                 _results = results;
649                 _stopableHandler = stopableHandler;
650                 _readAsElements = readAsElements;
651             }
652 
653             [MethodImpl(MethodImplOptions.AggressiveInlining)]
654             protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
                ↪  (_readAsElements != null && _readAsElements.Contains(link)) ||
                ↪  _linksInSequence.Contains(link);
655 
656             [MethodImpl(MethodImplOptions.AggressiveInlining)]
657             public bool FullMatch(LinkIndex sequenceToMatch)
658             {
659                 _filterPosition = 0;
660                 foreach (var part in Walk(sequenceToMatch))
661                 {
662                     if (!FullMatchCore(part))
663                     {
664                         break;
665                     }
666                 }
667                 return _filterPosition == _patternSequence.Count;
668             }
669 
670             [MethodImpl(MethodImplOptions.AggressiveInlining)]
671             private bool FullMatchCore(LinkIndex element)
672             {
673                 if (_filterPosition == _patternSequence.Count)
```

```
674                 {
675                     _filterPosition = -2; // Длиннее чем нужно
676                     return false;
677                 }
678                 if (_patternSequence[_filterPosition] != _links.Constants.Any
679                  && element != _patternSequence[_filterPosition])
680                 {
681                     _filterPosition = -1;
682                     return false; // Начинается/Продолжается иначе
683                 }
684                 _filterPosition++;
685                 return true;
686             }
687 
688             [MethodImpl(MethodImplOptions.AggressiveInlining)]
689             public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
690             {
691                 var sequenceToMatch = restrictions[_links.Constants.IndexPart];
692                 if (FullMatch(sequenceToMatch))
693                 {
694                     _results.Add(sequenceToMatch);
695                 }
696             }
697 
698             [MethodImpl(MethodImplOptions.AggressiveInlining)]
699             public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
700             {
701                 var sequenceToMatch = restrictions[_links.Constants.IndexPart];
702                 if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
703                 {
704                     return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
705                 }
706                 return _links.Constants.Continue;
707             }
708 
709             [MethodImpl(MethodImplOptions.AggressiveInlining)]
710             public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
711             {
712                 var sequenceToMatch = restrictions[_links.Constants.IndexPart];
713                 var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
714                 if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
                    ↪ _results.Add(sequenceToMatch))
715                 {
716                     return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
717                 }
718                 return _links.Constants.Continue;
719             }
720 
721             /// <remarks>
722             /// TODO: Add support for LinksConstants.Any
723             /// </remarks>
724             [MethodImpl(MethodImplOptions.AggressiveInlining)]
725             public bool PartialMatch(LinkIndex sequenceToMatch)
726             {
727                 _filterPosition = -1;
728                 foreach (var part in Walk(sequenceToMatch))
729                 {
730                     if (!PartialMatchCore(part))
731                     {
732                         break;
733                     }
734                 }
735                 return _filterPosition == _patternSequence.Count - 1;
736             }
737 
738             [MethodImpl(MethodImplOptions.AggressiveInlining)]
739             private bool PartialMatchCore(LinkIndex element)
740             {
741                 if (_filterPosition == (_patternSequence.Count - 1))
742                 {
743                     return false; // Нашлось
744                 }
745                 if (_filterPosition >= 0)
746                 {
747                     if (element == _patternSequence[_filterPosition + 1])
748                     {
749                         _filterPosition++;
750                     }
751                     else
```

```
                    {
                        _filterPosition = -1;
                    }
                }
                if (_filterPosition < 0)
                {
                    if (element == _patternSequence[0])
                    {
                        _filterPosition = 0;
                    }
                }
                return true; // Ищем дальше
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
            {
                if (PartialMatch(sequenceToMatch))
                {
                    _results.Add(sequenceToMatch);
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public LinkIndex HandlePartialMatched(IList<LinkIndex> restrictions)
            {
                var sequenceToMatch = restrictions[_links.Constants.IndexPart];
                if (PartialMatch(sequenceToMatch))
                {
                    return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
                }
                return _links.Constants.Continue;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
            {
                foreach (var sequenceToMatch in sequencesToMatch)
                {
                    if (PartialMatch(sequenceToMatch))
                    {
                        _results.Add(sequenceToMatch);
                    }
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
            ↪  sequencesToMatch)
            {
                foreach (var sequenceToMatch in sequencesToMatch)
                {
                    if (PartialMatch(sequenceToMatch))
                    {
                        _readAsElements.Add(sequenceToMatch);
                        _results.Add(sequenceToMatch);
                    }
                }
            }
        }

        #endregion
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences
{
    public static class SequencesExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink Create<TLink>(this ILinks<TLink> sequences, IList<TLink[]>
        ↪  groupedSequence)
```

```
        {
            var finalSequence = new TLink[groupedSequence.Count];
            for (var i = 0; i < finalSequence.Length; i++)
            {
                var part = groupedSequence[i];
                finalSequence[i] = part.Length == 1 ? part[0] :
                ↪  sequences.Create(part.ShiftRight());
            }
            return sequences.Create(finalSequence.ShiftRight());
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
        {
            var list = new List<TLink>();
            var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
            sequences.Each(filler.AddSkipFirstAndReturnConstant, new
            ↪  LinkAddress<TLink>(sequence));
            return list;
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Collections.Stacks;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Sequences.CriterionMatchers;
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Sequences
{
    public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
    ↪  ILinks<TLink> must contain GetConstants function.
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        public TLink SequenceMarkerLink
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseCascadeUpdate
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseCascadeDelete
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseIndex
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        } // TODO: Update Index on sequence update/delete.

        public bool UseSequenceMarker
        {
```

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

        public bool UseCompression
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseGarbageCollection
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public ISequenceIndex<TLink> Index
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public ISequenceWalker<TLink> Walker
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool ReadFullSequence
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        // TODO: Реализовать компактификацию при чтении
        //public bool EnforceSingleSequenceVersionOnRead { get; set; }
        //public bool UseRequestMarker { get; set; }
```

```
137         //public bool StoreRequestResults { get; set; }
138
139         [MethodImpl(MethodImplOptions.AggressiveInlining)]
140         public void InitOptions(ISynchronizedLinks<TLink> links)
141         {
142             if (UseSequenceMarker)
143             {
144                 if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
145                 {
146                     SequenceMarkerLink = links.CreatePoint();
147                 }
148                 else
149                 {
150                     if (!links.Exists(SequenceMarkerLink))
151                     {
152                         var link = links.CreatePoint();
153                         if (!_equalityComparer.Equals(link, SequenceMarkerLink))
154                         {
155                             throw new InvalidOperationException("Cannot recreate sequence marker
                                 ↪ link.");
156                         }
157                     }
158                 }
159                 if (MarkedSequenceMatcher == null)
160                 {
161                     MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                         ↪ SequenceMarkerLink);
162                 }
163             }
164             var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
165             if (UseCompression)
166             {
167                 if (LinksToSequenceConverter == null)
168                 {
169                     ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
170                     if (UseSequenceMarker)
171                     {
172                         totalSequenceSymbolFrequencyCounter = new
                             ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                             ↪ MarkedSequenceMatcher);
173                     }
174                     else
175                     {
176                         totalSequenceSymbolFrequencyCounter = new
                             ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
177                     }
178                     var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                         ↪ totalSequenceSymbolFrequencyCounter);
179                     var compressingConverter = new CompressingConverter<TLink>(links,
                         ↪ balancedVariantConverter, doubletFrequenciesCache);
180                     LinksToSequenceConverter = compressingConverter;
181                 }
182             }
183             else
184             {
185                 if (LinksToSequenceConverter == null)
186                 {
187                     LinksToSequenceConverter = balancedVariantConverter;
188                 }
189             }
190             if (UseIndex && Index == null)
191             {
192                 Index = new SequenceIndex<TLink>(links);
193             }
194             if (Walker == null)
195             {
196                 Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
197             }
198         }
199
200         [MethodImpl(MethodImplOptions.AggressiveInlining)]
201         public void ValidateOptions()
202         {
203             if (UseGarbageCollection && !UseSequenceMarker)
204             {
205                 throw new NotSupportedException("To use garbage collection UseSequenceMarker
                     ↪ option must be on.");
206             }
```

```
        }
    }
}
```

1.152 ./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.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.Sequences.Walkers
{
    public interface ISequenceWalker<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        IEnumerable<TLink> Walk(TLink sequence);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links, stack, isElement) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
        ↪  links.IsPartialPoint) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPop(TLink element) =>
        ↪  _links.GetSource(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPush(TLink element) =>
        ↪  _links.GetTarget(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLink> WalkContents(TLink element)
        {
            var links = _links;
            var parts = links.GetLink(element);
            var start = links.Constants.SourcePart;
            for (var i = parts.Count - 1; i >= start; i--)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

1.154 ./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs

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

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

//#define USEARRAYPOOL
#if USEARRAYPOOL
using Platform.Collections;
#endif

namespace Platform.Data.Doublets.Sequences.Walkers
{
```

```
14      public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
15      {
16          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
17
18          private readonly Func<TLink, bool> _isElement;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
            ↪  base(links) => _isElement = isElement;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
            ↪  _links.IsPartialPoint;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public TLink[] ToArray(TLink sequence)
31          {
32              var length = 1;
33              var array = new TLink[length];
34              array[0] = sequence;
35              if (_isElement(sequence))
36              {
37                  return array;
38              }
39              bool hasElements;
40              do
41              {
42                  length *= 2;
43  #if USEARRAYPOOL
44                  var nextArray = ArrayPool.Allocate<ulong>(length);
45  #else
46                  var nextArray = new TLink[length];
47  #endif
48                  hasElements = false;
49                  for (var i = 0; i < array.Length; i++)
50                  {
51                      var candidate = array[i];
52                      if (_equalityComparer.Equals(array[i], default))
53                      {
54                          continue;
55                      }
56                      var doubletOffset = i * 2;
57                      if (_isElement(candidate))
58                      {
59                          nextArray[doubletOffset] = candidate;
60                      }
61                      else
62                      {
63                          var links = _links;
64                          var link = links.GetLink(candidate);
65                          var linkSource = links.GetSource(link);
66                          var linkTarget = links.GetTarget(link);
67                          nextArray[doubletOffset] = linkSource;
68                          nextArray[doubletOffset + 1] = linkTarget;
69                          if (!hasElements)
70                          {
71                              hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
72                          }
73                      }
74                  }
75  #if USEARRAYPOOL
76                  if (array.Length > 1)
77                  {
78                      ArrayPool.Free(array);
79                  }
80  #endif
81                  array = nextArray;
82              }
83              while (hasElements);
84              var filledElementsCount = CountFilledElements(array);
85              if (filledElementsCount == array.Length)
86              {
87                  return array;
88              }
89              else
90              {
```

```
                return CopyFilledElements(array, filledElementsCount);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
        {
            var finalArray = new TLink[filledElementsCount];
            for (int i = 0, j = 0; i < array.Length; i++)
            {
                if (!_equalityComparer.Equals(array[i], default))
                {
                    finalArray[j] = array[i];
                    j++;
                }
            }
#if USEARRAYPOOL
                ArrayPool.Free(array);
#endif
            return finalArray;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static int CountFilledElements(TLink[] array)
        {
            var count = 0;
            for (var i = 0; i < array.Length; i++)
            {
                if (!_equalityComparer.Equals(array[i], default))
                {
                    count++;
                }
            }
            return count;
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links, stack, isElement) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
        ↪  stack, links.IsPartialPoint) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPop(TLink element) =>
        ↪  _links.GetTarget(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPush(TLink element) =>
        ↪  _links.GetSource(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLink> WalkContents(TLink element)
        {
            var parts = _links.GetLink(element);
            for (var i = _links.Constants.SourcePart; i < parts.Count; i++)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
```

```
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
    ↪ ISequenceWalker<TLink>
    {
        private readonly IStack<TLink> _stack;
        private readonly Func<TLink, bool> _isElement;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪ isElement) : base(links)
        {
            _stack = stack;
            _isElement = isElement;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
        ↪ stack, links.IsPartialPoint) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerable<TLink> Walk(TLink sequence)
        {
            _stack.Clear();
            var element = sequence;
            if (IsElement(element))
            {
                yield return element;
            }
            else
            {
                while (true)
                {
                    if (IsElement(element))
                    {
                        if (_stack.IsEmpty)
                        {
                            break;
                        }
                        element = _stack.Pop();
                        foreach (var output in WalkContents(element))
                        {
                            yield return output;
                        }
                        element = GetNextElementAfterPop(element);
                    }
                    else
                    {
                        _stack.Push(element);
                        element = GetNextElementAfterPush(element);
                    }
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract IEnumerable<TLink> WalkContents(TLink element);
    }
}
```

1.157 ./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));
    }
}
```

1.158 ./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;
        }
    }
}
```

1.159 ./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);

```

```
74          //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
            ↪  substitutedHandler, Unsync.Trigger);
75          //}
76      }
77  }
```

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

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

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

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

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

```
1   using System;
2   using System.Text;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Singletons;
6   using Platform.Data.Doublets.Unicode;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets
11  {
12      public static class UInt64LinksExtensions
13      {
14          public static readonly LinksConstants<ulong> Constants =
            ↪  Default<LinksConstants<ulong>>.Instance;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
21          {
22              if (sequence == null)
23              {
24                  return false;
```

```
            }
            var constants = links.Constants;
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] == constants.Any)
                {
                    return true;
                }
            }
            return false;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
        ↪  Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
        ↪  false)
        {
            var sb = new StringBuilder();
            var visited = new HashSet<ulong>();
            links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
            ↪  innerSb.Append(link.Index), renderIndex, renderDebug);
            return sb.ToString();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
        ↪  Func<Link<ulong>, bool> isElement, Action<StringBuilder, Link<ulong>> appendElement,
        ↪  bool renderIndex = false, bool renderDebug = false)
        {
            var sb = new StringBuilder();
            var visited = new HashSet<ulong>();
            links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
            ↪  renderDebug);
            return sb.ToString();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
        ↪  HashSet<ulong> visited, ulong linkIndex, Func<Link<ulong>, bool> isElement,
        ↪  Action<StringBuilder, Link<ulong>> appendElement, bool renderIndex = false, bool
        ↪  renderDebug = false)
        {
            if (sb == null)
            {
                throw new ArgumentNullException(nameof(sb));
            }
            if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
            ↪  Constants.Itself)
            {
                return;
            }
            if (links.Exists(linkIndex))
            {
                if (visited.Add(linkIndex))
                {
                    sb.Append('(');
                    var link = new Link<ulong>(links.GetLink(linkIndex));
                    if (renderIndex)
                    {
                        sb.Append(link.Index);
                        sb.Append(':');
                    }
                    if (link.Source == link.Index)
                    {
                        sb.Append(link.Index);
                    }
                    else
                    {
                        var source = new Link<ulong>(links.GetLink(link.Source));
                        if (isElement(source))
                        {
                            appendElement(sb, source);
                        }
                        else
                        {
                            links.AppendStructure(sb, visited, source.Index, isElement,
                            ↪  appendElement, renderIndex);
                        }
```

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

1.163 ./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)]
```

```
110             public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
                ↪ {After}";
111
112             [MethodImpl(MethodImplOptions.AggressiveInlining)]
113             public override bool Equals(object obj) => obj is Transition transition ?
                ↪ Equals(transition) : false;
114
115             [MethodImpl(MethodImplOptions.AggressiveInlining)]
116             public override int GetHashCode() => (TransactionId, Before, After,
                ↪ Timestamp).GetHashCode();
117
118             [MethodImpl(MethodImplOptions.AggressiveInlining)]
119             public bool Equals(Transition other) => TransactionId == other.TransactionId &&
                ↪ Before == other.Before && After == other.After && Timestamp == other.Timestamp;
120
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
191             [MethodImpl(MethodImplOptions.AggressiveInlining)]
192             private void Revert()
193             {
194                 EnsureTransactionAllowsWriteOperations(this);
195                 var transitionsToRevert = new Transition[_transitions.Count];
196                 _transitions.CopyTo(transitionsToRevert, 0);
197                 for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
198                 {
199                     _layer.RevertTransition(transitionsToRevert[i]);
200                 }
201                 IsReverted = true;
202             }
203
204             [MethodImpl(MethodImplOptions.AggressiveInlining)]
205             public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
                ↪  Transaction transaction)
206             {
207                 layer._currentTransactionId = layer._lastCommitedTransactionId + 1;
208                 layer._currentTransactionTransitions = transaction._transitions;
209                 layer._currentTransaction = transaction;
210             }
211
212             [MethodImpl(MethodImplOptions.AggressiveInlining)]
213             public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
214             {
215                 if (transaction.IsReverted)
216                 {
217                     throw new InvalidOperationException("Transation is reverted.");
218                 }
219                 if (transaction.IsCommitted)
220                 {
221                     throw new InvalidOperationException("Transation is commited.");
222                 }
223             }
224
225             [MethodImpl(MethodImplOptions.AggressiveInlining)]
226             protected override void Dispose(bool manual, bool wasDisposed)
227             {
228                 if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
229                 {
230                     if (!IsCommitted && !IsReverted)
231                     {
232                         Revert();
233                     }
234                     _layer.ResetCurrentTransation();
235                 }
236             }
237         }
238
239         public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
240
241         private readonly string _logAddress;
242         private readonly FileStream _log;
243         private readonly Queue<Transition> _transitions;
244         private readonly UniqueTimestampFactory _uniqueTimestampFactory;
245         private Task _transitionsPusher;
246         private Transition _lastCommitedTransition;
247         private ulong _currentTransactionId;
248         private Queue<Transition> _currentTransactionTransitions;
249         private Transaction _currentTransaction;
250         private ulong _lastCommitedTransactionId;
251
252         [MethodImpl(MethodImplOptions.AggressiveInlining)]
253         public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
254             : base(links)
```

```
255         {
256             if (string.IsNullOrWhiteSpace(logAddress))
257             {
258                 throw new ArgumentNullException(nameof(logAddress));
259             }
260             // В первой строке файла хранится последняя закоммиченную транзакцию.
261             // При запуске это используется для проверки удачного закрытия файла лога.
262             // In the first line of the file the last committed transaction is stored.
263             // On startup, this is used to check that the log file is successfully closed.
264             var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
265             var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
266             if (!lastCommitedTransition.Equals(lastWrittenTransition))
267             {
268                 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             {
```

```
                Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
            }
            var transitions = GetCurrentTransitions();
            transitions.Enqueue(transition);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void RevertTransition(Transition transition)
        {
            if (transition.After.IsNull()) // Revert Deletion with Creation
            {
                _links.Create();
            }
            else if (transition.Before.IsNull()) // Revert Creation with Deletion
            {
                _links.Delete(transition.After.Index);
            }
            else // Revert Update
            {
                _links.Update(new[] { transition.After.Index, transition.Before.Source,
                ↪  transition.Before.Target });
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ResetCurrentTransation()
        {
            _currentTransactionId = 0;
            _currentTransactionTransitions = null;
            _currentTransaction = null;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void PushTransitions()
        {
            if (_log == null || _transitions == null)
            {
                return;
            }
            for (var i = 0; i < _transitions.Count; i++)
            {
                var transition = _transitions.Dequeue();

                _log.Write(transition);
                _lastCommitedTransition = transition;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void TransitionsPusher()
        {
            while (!Disposable.IsDisposed && _transitionsPusher != null)
            {
                Thread.Sleep(DefaultPushDelay);
                PushTransitions();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Transaction BeginTransaction() => new Transaction(this);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void DisposeTransitions()
        {
            try
            {
                var pusher = _transitionsPusher;
                if (pusher != null)
                {
                    _transitionsPusher = null;
                    pusher.Wait();
                }
                if (_transitions != null)
                {
                    PushTransitions();
                }
                _log.DisposeIfPossible();
                FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
            }
```

```
            catch (Exception ex)
            {
                ex.Ignore();
            }
        }

        #region DisposalBase

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

        #endregion
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Unicode
{
    public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<char, TLink>
    {
        private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
        ↪ UncheckedConverter<char, TLink>.Default;

        private readonly IConverter<TLink> _addressToNumberConverter;
        private readonly TLink _unicodeSymbolMarker;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
        ↪ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
        {
            _addressToNumberConverter = addressToNumberConverter;
            _unicodeSymbolMarker = unicodeSymbolMarker;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(char source)
        {
            var unaryNumber =
            ↪ _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
            return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
        }
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Indexes;

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

namespace Platform.Data.Doublets.Unicode
{
    public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<string, TLink>
    {
        private readonly IConverter<string, IList<TLink>> _stringToUnicodeSymbolListConverter;
        private readonly IConverter<IList<TLink>, TLink> _unicodeSymbolListToSequenceConverter;

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

```
        }

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Unicode
{
    public class StringToUnicodeSymbolsListConverter<TLink> : IConverter<string, IList<TLink>>
    {
        private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;

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

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

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

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

```
using System.Text;
using Platform.Data.Sequences;

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

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeMap
    {
        public static readonly ulong FirstCharLink = 1;
        public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
        public static readonly ulong MapSize = 1 + char.MaxValue;

        private readonly ILinks<ulong> _links;
        private bool _initialized;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeMap(ILinks<ulong> links) => _links = links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static UnicodeMap InitNew(ILinks<ulong> links)
        {
            var map = new UnicodeMap(links);
            map.Init();
            return map;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Init()
        {
            if (_initialized)
            {
                return;
            }
            _initialized = true;
            var firstLink = _links.CreatePoint();
            if (firstLink != FirstCharLink)
            {
                _links.Delete(firstLink);
            }
            else
            {
                for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
                {
                    // From NIL to It (NIL -> Character) transformation meaning, (or infinite amount of NIL characters before actual Character)
                    var createdLink = _links.CreatePoint();
                    _links.Update(createdLink, firstLink, createdLink);
                    if (createdLink != i)
                    {
                        throw new InvalidOperationException("Unable to initialize UTF 16 table.");
                    }
                }
            }
        }

        // 0 - null link
        // 1 - nil character (0 character)
        // ...
        // 65536 (0(1) + 65535 = 65536 possible values)

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong FromCharToLink(char character) => (ulong)character + 1;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static char FromLinkToChar(ulong link) => (char)(link - 1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsCharLink(ulong link) => link <= MapSize;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string FromLinksToString(IList<ulong> linksList)
        {
            var sb = new StringBuilder();
            for (int i = 0; i < linksList.Count; i++)
            {
                sb.Append(FromLinkToChar(linksList[i]));
            }
```

```
            return sb.ToString();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
        {
            var sb = new StringBuilder();
            if (links.Exists(link))
            {
                StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
                    x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                    element =>
                    {
                        sb.Append(FromLinkToChar(element));
                        return true;
                    });
            }
            return sb.ToString();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
        chars.Length);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong[] FromCharsToLinkArray(char[] chars, int count)
        {
            // char array to ulong array
            var linksSequence = new ulong[count];
            for (var i = 0; i < count; i++)
            {
                linksSequence[i] = FromCharToLink(chars[i]);
            }
            return linksSequence;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong[] FromStringToLinkArray(string sequence)
        {
            // char array to ulong array
            var linksSequence = new ulong[sequence.Length];
            for (var i = 0; i < sequence.Length; i++)
            {
                linksSequence[i] = FromCharToLink(sequence[i]);
            }
            return linksSequence;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
        {
            var result = new List<ulong[]>();
            var offset = 0;
            while (offset < sequence.Length)
            {
                var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
                var relativeLength = 1;
                var absoluteLength = offset + relativeLength;
                while (absoluteLength < sequence.Length &&
                       currentCategory ==
                       CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
                {
                    relativeLength++;
                    absoluteLength++;
                }
                // char array to ulong array
                var innerSequence = new ulong[relativeLength];
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = FromCharToLink(sequence[i]);
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
158         public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
159         {
160             var result = new List<ulong[]>();
161             var offset = 0;
162             while (offset < array.Length)
163             {
164                 var relativeLength = 1;
165                 if (array[offset] <= LastCharLink)
166                 {
167                     var currentCategory =
                        ↪ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
168                     var absoluteLength = offset + relativeLength;
169                     while (absoluteLength < array.Length &&
170                            array[absoluteLength] <= LastCharLink &&
171                            currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
                               ↪ array[absoluteLength])))
172                     {
173                         relativeLength++;
174                         absoluteLength++;
175                     }
176                 }
177                 else
178                 {
179                     var absoluteLength = offset + relativeLength;
180                     while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
181                     {
182                         relativeLength++;
183                         absoluteLength++;
184                     }
185                 }
186                 // copy array
187                 var innerSequence = new ulong[relativeLength];
188                 var maxLength = offset + relativeLength;
189                 for (var i = offset; i < maxLength; i++)
190                 {
191                     innerSequence[i - offset] = array[i];
192                 }
193                 result.Add(innerSequence);
194                 offset += relativeLength;
195             }
196             return result;
197         }
198     }
199 }
```

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

```
1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Converters;
5  using Platform.Data.Doublets.Sequences.Walkers;
6  using System.Text;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
       ↪ IConverter<TLink, string>
13     {
14         private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
15         private readonly ISequenceWalker<TLink> _sequenceWalker;
16         private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
           ↪ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
           ↪ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
20         {
21             _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
22             _sequenceWalker = sequenceWalker;
23             _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public string Convert(TLink source)
28         {
29             if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
30             {
```

```
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                ↪  not a unicode sequence.");
            }
            var sequence = _links.GetSource(source);
            var sb = new StringBuilder();
            foreach(var character in _sequenceWalker.Walk(sequence))
            {
                sb.Append(_unicodeSymbolToCharConverter.Convert(character));
            }
            return sb.ToString();
        }
    }
}
```

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

```
using System;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Converters;

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

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<TLink, char>
    {
        private static readonly UncheckedConverter<TLink, char> _addressToCharConverter =
        ↪  UncheckedConverter<TLink, char>.Default;

        private readonly IConverter<TLink> _numberToAddressConverter;
        private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
        ↪  numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
        ↪  base(links)
        {
            _numberToAddressConverter = numberToAddressConverter;
            _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public char Convert(TLink source)
        {
            if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                ↪  not a unicode symbol.");
            }
            return _addressToCharConverter.Convert(_numberToAddressConverter.Convert(_links.GetS
            ↪  ource(source)));
        }
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Indexes;

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

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSymbolsListToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<IList<TLink>, TLink>
    {
        private readonly ISequenceIndex<TLink> _index;
        private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
        private readonly TLink _unicodeSequenceMarker;

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

```
            _listToSequenceLinkConverter = listToSequenceLinkConverter;
            _unicodeSequenceMarker = unicodeSequenceMarker;
        }

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

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

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

```
using System;
using Xunit;
using Platform.Reflection;
using Platform.Memory;
using Platform.Scopes;
using Platform.Data.Doublets.Memory.United.Generic;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class GenericLinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using<byte>(links => links.TestCRUDOperations());
            Using<ushort>(links => links.TestCRUDOperations());
            Using<uint>(links => links.TestCRUDOperations());
            Using<ulong>(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            Using<byte>(links => links.TestRawNumbersCRUDOperations());
            Using<ushort>(links => links.TestRawNumbersCRUDOperations());
            Using<uint>(links => links.TestRawNumbersCRUDOperations());
            Using<ulong>(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
            ↪  MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
            ↪  implementation of tree cuts out 5 bits from the address space.
            Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
            ↪  stMultipleRandomCreationsAndDeletions(100));
            Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
            ↪  MultipleRandomCreationsAndDeletions(100));
            Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
            ↪  tMultipleRandomCreationsAndDeletions(100));
        }

        private static void Using<TLink>(Action<ILinks<TLink>> action)
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  UnitedMemoryLinks<TLink>>>())
            {
                action(scope.Use<ILinks<TLink>>());
            }
        }
    }
}
```

1.172 ./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs

```
using Xunit;

```

```
namespace Platform.Data.Doublets.Tests
{
    public class ILinksExtensionsTests
    {
        [Fact]
        public void FormatTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var link = links.Create();
                var linkString = links.Format(link);
                Assert.Equal("(1: 1 1)", linkString);
            }
        }
    }
}
```

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

```
using Xunit;

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

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

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

1.174 ./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs

```
using System;
using System.Linq;
using Xunit;
using Platform.Collections.Stacks;
using Platform.Collections.Arrays;
using Platform.Memory;
using Platform.Data.Numbers.Raw;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Doublets.Numbers.Unary;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Memory.United.Specific;
using Platform.Data.Doublets.Memory;

namespace Platform.Data.Doublets.Tests
{
    public static class OptimalVariantSequenceTests
    {
        private static readonly string _sequenceExample = "зеленела зелёная зелень";
        private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
Et malesuada fames ac turpis egestas sed.
Eget velit aliquet sagittis id consectetur purus.
Dignissim cras tincidunt lobortis feugiat vivamus.
Vitae aliquet nec ullamcorper sit.
Lectus quam id leo in vitae.
Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
Integer eget aliquet nibh praesent tristique.
Vitae congue eu consequat ac felis donec et odio.
```

```
Tristique et egestas quis ipsum suspendisse.
Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
Imperdiet proin fermentum leo vel orci.
In ante metus dictum at tempor commodo.
Nisi lacus sed viverra tellus in.
Quam vulputate dignissim suspendisse in.
Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
Gravida cum sociis natoque penatibus et magnis dis parturient.
Risus quis varius quam quisque id diam.
Congue nisi vitae suscipit tellus mauris a diam maecenas.
Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
Pharetra vel turpis nunc eget lorem dolor sed viverra.
Mattis pellentesque id nibh tortor id aliquet.
Purus non enim praesent elementum facilisis leo vel.
Etiam sit amet nisl purus in mollis nunc sed.
Tortor at auctor urna nunc id cursus metus aliquam.
Volutpat odio facilisis mauris sit amet.
Turpis egestas pretium aenean pharetra magna ac placerat.
Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
Porttitor leo a diam sollicitudin tempor id eu.
Volutpat sed cras ornare arcu dui.
Ut aliquam purus sit amet luctus venenatis lectus magna.
Aliquet risus feugiat in ante metus dictum at.
Mattis nunc sed blandit libero.
Elit pellentesque habitant morbi tristique senectus et netus.
Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
Diam donec adipiscing tristique risus nec feugiat.
Pulvinar mattis nunc sed blandit libero volutpat.
Cras fermentum odio eu feugiat pretium nibh ipsum.
In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
A iaculis at erat pellentesque.
Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
Eget lorem dolor sed viverra ipsum nunc.
Leo a diam sollicitudin tempor id eu.
Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";

        [Fact]
        public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;
                var constants = links.Constants;

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

                var meaningRoot = links.CreatePoint();
                var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
                var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
                var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                    constants.Itself);

                var unaryNumberToAddressConverter = new
                    UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
                var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
                var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    frequencyMarker, unaryOne, unaryNumberIncrementer);
                var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                    frequencyPropertyMarker, frequencyMarker);
                var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                    frequencyPropertyOperator, frequencyIncrementer);
                var linkToItsFrequencyNumberConverter = new
                    LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    unaryNumberToAddressConverter);
                var sequenceToItsLocalElementLevelsConverter = new
                    SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    sequenceToItsLocalElementLevelsConverter);

                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                    Walker = new LeveledSequenceWalker<ulong>(links) });

                ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                    index, optimalVariantConverter);
```

```
107             }
108         }
109 
110         [Fact]
111         public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
112         {
113             using (var scope = new TempLinksTestScope(useSequences: false))
114             {
115                 var links = scope.Links;
116 
117                 links.UseUnicode();
118 
119                 var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
120 
121                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
122 
123                 var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                    ↪  totalSequenceSymbolFrequencyCounter);
124 
125                 var index = new
                    ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
126                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache);
127 
128                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
129                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
130 
131                 var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                    ↪  Walker = new LeveledSequenceWalker<ulong>(links) });
132 
133                 ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                    ↪  index, optimalVariantConverter);
134             }
135         }
136 
137         private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
            ↪  SequenceToItsLocalElementLevelsConverter<ulong>
            ↪  sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
            ↪  OptimalVariantConverter<ulong> optimalVariantConverter)
138         {
139             index.Add(sequence);
140 
141             var optimalVariant = optimalVariantConverter.Convert(sequence);
142 
143             var readSequence1 = sequences.ToList(optimalVariant);
144 
145             Assert.True(sequence.SequenceEqual(readSequence1));
146         }
147 
148         [Fact]
149         public static void SavedSequencesOptimizationTest()
150         {
151             LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
                ↪  (long.MaxValue + 1UL, ulong.MaxValue));
152 
153             using (var memory = new HeapResizableDirectMemory())
154             using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
                ↪  UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, IndexTreeType.Default))
155             {
156                 var links = new UInt64Links(disposableLinks);
157 
158                 var root = links.CreatePoint();
159 
160                 //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
161                 var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
162 
163                 var unicodeSymbolMarker = links.GetOrCreate(root,
                    ↪  addressToNumberConverter.Convert(1));
164                 var unicodeSequenceMarker = links.GetOrCreate(root,
                    ↪  addressToNumberConverter.Convert(2));
165 
166                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
167                 var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                    ↪  totalSequenceSymbolFrequencyCounter);
```

```
                var index = new
                ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
                var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                ↪  ncyNumberConverter<ulong>(linkFrequenciesCache);
                var sequenceToItsLocalElementLevelsConverter = new
                ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                ↪  linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                ↪  sequenceToItsLocalElementLevelsConverter);

                var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
                ↪  (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));

                var unicodeSequencesOptions = new SequencesOptions<ulong>()
                {
                    UseSequenceMarker = true,
                    SequenceMarkerLink = unicodeSequenceMarker,
                    UseIndex = true,
                    Index = index,
                    LinksToSequenceConverter = optimalVariantConverter,
                    Walker = walker,
                    UseGarbageCollection = true
                };

                var unicodeSequences = new Sequences.Sequences(new
                ↪  SynchronizedLinks<ulong>(links), unicodeSequencesOptions);

                // Create some sequences
                var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
                ↪  StringSplitOptions.RemoveEmptyEntries);
                var arrays = strings.Select(x => x.Select(y =>
                ↪  addressToNumberConverter.Convert(y)).ToArray()).ToArray();
                for (int i = 0; i < arrays.Length; i++)
                {
                    unicodeSequences.Create(arrays[i].ShiftRight());
                }

                var linksCountAfterCreation = links.Count();

                // get list of sequences links
                // for each sequence link
                //   create new sequence version
                //   if new sequence is not the same as sequence link
                //     delete sequence link
                //     collect garbadge
                unicodeSequences.CompactAll();

                var linksCountAfterCompactification = links.Count();

                Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
            }
        }
    }
}
```

1.175 ./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences;

namespace Platform.Data.Doublets.Tests
{
    public static class ReadSequenceTests
    {
        [Fact]
        public static void ReadSequenceTest()
        {
            const long sequenceLength = 2000;

            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;
                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong> {
                ↪  Walker = new LeveledSequenceWalker<ulong>(links) });
```

```

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

                var sw1 = Stopwatch.StartNew();
                var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();

                var sw2 = Stopwatch.StartNew();
                var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();

                var sw3 = Stopwatch.StartNew();
                var readSequence2 = new List<ulong>();
                SequenceWalker.WalkRight(balancedVariant,
                                         links.GetSource,
                                         links.GetTarget,
                                         links.IsPartialPoint,
                                         readSequence2.Add);
                sw3.Stop();

                Assert.True(sequence.SequenceEqual(readSequence1));

                Assert.True(sequence.SequenceEqual(readSequence2));

                // Assert.True(sw2.Elapsed < sw3.Elapsed);

                Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
                ↪  {sw2.Elapsed}");

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }
    }
}
```

1.176 ./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);
        }
    }
}
```

1.177 ./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);
            }
        }
    }
}
```

1.178 ./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Collections;
using Platform.Collections.Arrays;
using Platform.Random;
using Platform.IO;
using Platform.Singletons;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Unicode;

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

        static SequencesTests()
        {
            // Trigger static constructor to not mess with perfomance measurements
            _ = BitString.GetBitMaskFromIndex(1);
        }

        [Fact]
        public static void CreateAllVariantsTest()
        {
            const long sequenceLength = 8;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var sw1 = Stopwatch.StartNew();
                var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();

                var sw2 = Stopwatch.StartNew();
                var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();

                Assert.True(results1.Count > results2.Length);
                Assert.True(sw1.Elapsed > sw2.Elapsed);

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }

                Assert.True(links.Count() == 0);
            }
        }

        //[Fact]
```

```
        //public void CUDTest()
        //{
        //    var tempFilename = Path.GetTempFileName();

        //    const long sequenceLength = 8;

        //    const ulong itself = LinksConstants.Itself;

        //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
        ↪  DefaultLinksSizeStep))
        //    using (var links = new Links(memoryAdapter))
        //    {
        //        var sequence = new ulong[sequenceLength];
        //        for (var i = 0; i < sequenceLength; i++)
        //            sequence[i] = links.Create(itself, itself);

        //        SequencesOptions o = new SequencesOptions();

        // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
        //        o.

        //        var sequences = new Sequences(links);

        //        var sw1 = Stopwatch.StartNew();
        //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();

        //        var sw2 = Stopwatch.StartNew();
        //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();

        //        Assert.True(results1.Count > results2.Length);
        //        Assert.True(sw1.Elapsed > sw2.Elapsed);

        //        for (var i = 0; i < sequenceLength; i++)
        //            links.Delete(sequence[i]);
        //    }

        //    File.Delete(tempFilename);
        //}

        [Fact]
        public static void AllVariantsSearchTest()
        {
            const long sequenceLength = 8;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();

                //for (int i = 0; i < createResults.Length; i++)
                //    sequences.Create(createResults[i]);

                var sw0 = Stopwatch.StartNew();
                var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();

                var sw1 = Stopwatch.StartNew();
                var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();

                var sw2 = Stopwatch.StartNew();
                var searchResults2 = sequences.Each1(sequence); sw2.Stop();

                var sw3 = Stopwatch.StartNew();
                var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();

                var intersection0 = createResults.Intersect(searchResults0).ToList();
                Assert.True(intersection0.Count == searchResults0.Count);
                Assert.True(intersection0.Count == createResults.Length);

                var intersection1 = createResults.Intersect(searchResults1).ToList();
                Assert.True(intersection1.Count == searchResults1.Count);
                Assert.True(intersection1.Count == createResults.Length);

```

```
143                 var intersection2 = createResults.Intersect(searchResults2).ToList();
144                 Assert.True(intersection2.Count == searchResults2.Count);
145                 Assert.True(intersection2.Count == createResults.Length);
146
147                 var intersection3 = createResults.Intersect(searchResults3).ToList();
148                 Assert.True(intersection3.Count == searchResults3.Count);
149                 Assert.True(intersection3.Count == createResults.Length);
150
151                 for (var i = 0; i < sequenceLength; i++)
152                 {
153                     links.Delete(sequence[i]);
154                 }
155             }
156         }
157
158         [Fact]
159         public static void BalancedVariantSearchTest()
160         {
161             const long sequenceLength = 200;
162
163             using (var scope = new TempLinksTestScope(useSequences: true))
164             {
165                 var links = scope.Links;
166                 var sequences = scope.Sequences;
167
168                 var sequence = new ulong[sequenceLength];
169                 for (var i = 0; i < sequenceLength; i++)
170                 {
171                     sequence[i] = links.Create();
172                 }
173
174                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
175
176                 var sw1 = Stopwatch.StartNew();
177                 var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
178
179                 var sw2 = Stopwatch.StartNew();
180                 var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
181
182                 var sw3 = Stopwatch.StartNew();
183                 var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
184
185                 // На количестве в 200 элементов это будет занимать вечность
186                 //var sw4 = Stopwatch.StartNew();
187                 //var searchResults4 = sequences.Each(sequence); sw4.Stop();
188
189                 Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
190
191                 Assert.True(searchResults3.Count == 1 && balancedVariant ==
                    ↪ searchResults3.First());
192
193                 //Assert.True(sw1.Elapsed < sw2.Elapsed);
194
195                 for (var i = 0; i < sequenceLength; i++)
196                 {
197                     links.Delete(sequence[i]);
198                 }
199             }
200         }
201
202         [Fact]
203         public static void AllPartialVariantsSearchTest()
204         {
205             const long sequenceLength = 8;
206
207             using (var scope = new TempLinksTestScope(useSequences: true))
208             {
209                 var links = scope.Links;
210                 var sequences = scope.Sequences;
211
212                 var sequence = new ulong[sequenceLength];
213                 for (var i = 0; i < sequenceLength; i++)
214                 {
215                     sequence[i] = links.Create();
216                 }
217
218                 var createResults = sequences.CreateAllVariants2(sequence);
219
220                 //var createResultsStrings = createResults.Select(x => x + ": " +
                    ↪ sequences.FormatSequence(x)).ToList();
```

```
            //Global.Trash = createResultsStrings;

            var partialSequence = new ulong[sequenceLength - 2];

            Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);

            var sw1 = Stopwatch.StartNew();
            var searchResults1 =
            ↪  sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();

            var sw2 = Stopwatch.StartNew();
            var searchResults2 =
            ↪  sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();

            //var sw3 = Stopwatch.StartNew();
            //var searchResults3 =
            ↪  sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();

            var sw4 = Stopwatch.StartNew();
            var searchResults4 =
            ↪  sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();

            //Global.Trash = searchResults3;

            //var searchResults1Strings = searchResults1.Select(x => x + ": " +
            ↪  sequences.FormatSequence(x)).ToList();
            //Global.Trash = searchResults1Strings;

            var intersection1 = createResults.Intersect(searchResults1).ToList();
            Assert.True(intersection1.Count == createResults.Length);

            var intersection2 = createResults.Intersect(searchResults2).ToList();
            Assert.True(intersection2.Count == createResults.Length);

            var intersection4 = createResults.Intersect(searchResults4).ToList();
            Assert.True(intersection4.Count == createResults.Length);

            for (var i = 0; i < sequenceLength; i++)
            {
                links.Delete(sequence[i]);
            }
        }
    }

    [Fact]
    public static void BalancedPartialVariantsSearchTest()
    {
        const long sequenceLength = 200;

        using (var scope = new TempLinksTestScope(useSequences: true))
        {
            var links = scope.Links;
            var sequences = scope.Sequences;

            var sequence = new ulong[sequenceLength];
            for (var i = 0; i < sequenceLength; i++)
            {
                sequence[i] = links.Create();
            }

            var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

            var balancedVariant = balancedVariantConverter.Convert(sequence);

            var partialSequence = new ulong[sequenceLength - 2];

            Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);

            var sw1 = Stopwatch.StartNew();
            var searchResults1 =
            ↪  sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();

            var sw2 = Stopwatch.StartNew();
            var searchResults2 =
            ↪  sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();

            Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);

            Assert.True(searchResults2.Count == 1 && balancedVariant ==
            ↪  searchResults2.First());
```

```
293
294                 for (var i = 0; i < sequenceLength; i++)
295                 {
296                     links.Delete(sequence[i]);
297                 }
298             }
299         }
300
301         [Fact(Skip = "Correct implementation is pending")]
302         public static void PatternMatchTest()
303         {
304             var zeroOrMany = Sequences.Sequences.ZeroOrMany;
305
306             using (var scope = new TempLinksTestScope(useSequences: true))
307             {
308                 var links = scope.Links;
309                 var sequences = scope.Sequences;
310
311                 var e1 = links.Create();
312                 var e2 = links.Create();
313
314                 var sequence = new[]
315                 {
316                     e1, e2, e1, e2 // mama / papa
317                 };
318
319                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
320
321                 var balancedVariant = balancedVariantConverter.Convert(sequence);
322
323                 // 1: [1]
324                 // 2: [2]
325                 // 3: [1,2]
326                 // 4: [1,2,1,2]
327
328                 var doublet = links.GetSource(balancedVariant);
329
330                 var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
331
332                 Assert.True(matchedSequences1.Count == 0);
333
334                 var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
335
336                 Assert.True(matchedSequences2.Count == 0);
337
338                 var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
339
340                 Assert.True(matchedSequences3.Count == 0);
341
342                 var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
343
344                 Assert.Contains(doublet, matchedSequences4);
345                 Assert.Contains(balancedVariant, matchedSequences4);
346
347                 for (var i = 0; i < sequence.Length; i++)
348                 {
349                     links.Delete(sequence[i]);
350                 }
351             }
352         }
353
354         [Fact]
355         public static void IndexTest()
356         {
357             using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
                 ↪ true }, useSequences: true))
358             {
359                 var links = scope.Links;
360                 var sequences = scope.Sequences;
361                 var index = sequences.Options.Index;
362
363                 var e1 = links.Create();
364                 var e2 = links.Create();
365
366                 var sequence = new[]
367                 {
368                     e1, e2, e1, e2 // mama / papa
369                 };
370
371                 Assert.False(index.MightContain(sequence));
```

```
372
373                 index.Add(sequence);
374
375                 Assert.True(index.MightContain(sequence));
376             }
377         }
378
379         /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87%D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
380         private static readonly string _exampleText =
381             @"([english
382 version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
383
384 Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там где есть место для нового начала? Разве пустота это не характеристика пространства? Пространство это то, что можно чем-то наполнить?
385
386 [![чёрное пространство, белое пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png ""чёрное пространство, белое пространство"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png)
387
388 Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?
389
390 [![чёрное пространство, чёрная точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png ""чёрное пространство, чёрная точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)
391
392 А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть так? Инверсия? Отражение? Сумма?
393
394 [![белая точка, чёрная точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png ""белая точка, чёрная точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)
395
396 А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой? Гранью? Разделителем? Единицей?
397
398 [![две белые точки, чёрная вертикальная линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png ""две белые точки, чёрная вертикальная линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)
399
400 Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец? Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если у него нет размера? Будет ли круг точкой? Точка состоящая из точек?
401
402 [![белая вертикальная линия, чёрный круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png ""белая вертикальная линия, чёрный круг"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)
403
404 Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально? Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли элементарная единица смысла?
405
406 [![белый круг, чёрная горизонтальная линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png ""белый круг, чёрная горизонтальная линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)
407
408 Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла ""соединить, связать"", есть ещё и смысл направления ""от начала к концу""? От предка к потомку? От родителя к ребёнку? От общего к частному?
409
410 [![белая горизонтальная линия, чёрная горизонтальная стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png ""белая горизонтальная линия, чёрная горизонтальная стрелка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)
411
```

```
Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два объекта, как бы это выглядело?

[![белая связь, чёрная направленная связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png ""белая связь, чёрная направленная связь"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)

Допустим у нас есть смысл ""связать"" и смысл ""направления"", много ли это нам даёт? Много ли вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие? Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в его конечном состоянии, если конечно конец определён направлением?

[![белая обычная и направленная связи, чёрная типизированная связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png ""белая обычная и направленная связи, чёрная типизированная связь"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)

А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри? Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?

[![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная связь с рекурсивной внутренней структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png ""белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная связь с рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)

На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом рекурсии или фрактала?

[![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная типизированная связь с двойной рекурсивной внутренней структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png ""белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная типизированная связь с двойной рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)

Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы? Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?

[![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png ""белая обычная и направленная связи со структурой из 8 цветных элементов последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последовательности"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)

...

[![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif)";

        private static readonly string _exampleLoremIpsumText =
            @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.";

        [Fact]
        public static void CompressionTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var e1 = links.Create();
                var e2 = links.Create();

                var sequence = new[]
                {
                    e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
```

```
453                 };
454
455                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
456                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
457                 var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
458                 var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
                    ↪  balancedVariantConverter, doubletFrequenciesCache);
459
460                 var compressedVariant = compressingConverter.Convert(sequence);
461
462                 // 1: [1]       (1->1) point
463                 // 2: [2]       (2->2) point
464                 // 3: [1,2]     (1->2) doublet
465                 // 4: [1,2,1,2] (3->3) doublet
466
467                 Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
468                 Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
469                 Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
470                 Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
471
472                 var source = _constants.SourcePart;
473                 var target = _constants.TargetPart;
474
475                 Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
476                 Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
477                 Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
478                 Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
479
480                 // 4 - length of sequence
481                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
                    ↪  == sequence[0]);
482                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
                    ↪  == sequence[1]);
483                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
                    ↪  == sequence[2]);
484                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
                    ↪  == sequence[3]);
485             }
486         }
487
488         [Fact]
489         public static void CompressionEfficiencyTest()
490         {
491             var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
                ↪  StringSplitOptions.RemoveEmptyEntries);
492             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
493             var totalCharacters = arrays.Select(x => x.Length).Sum();
494
495             using (var scope1 = new TempLinksTestScope(useSequences: true))
496             using (var scope2 = new TempLinksTestScope(useSequences: true))
497             using (var scope3 = new TempLinksTestScope(useSequences: true))
498             {
499                 scope1.Links.Unsync.UseUnicode();
500                 scope2.Links.Unsync.UseUnicode();
501                 scope3.Links.Unsync.UseUnicode();
502
503                 var balancedVariantConverter1 = new
                    ↪  BalancedVariantConverter<ulong>(scope1.Links.Unsync);
504                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
505                 var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
506                 var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
                    ↪  balancedVariantConverter1, linkFrequenciesCache1,
                    ↪  doInitialFrequenciesIncrement: false);
507
508                 //var compressor2 = scope2.Sequences;
509                 var compressor3 = scope3.Sequences;
510
511                 var constants = Default<LinksConstants<ulong>>.Instance;
512
513                 var sequences = compressor3;
514                 //var meaningRoot = links.CreatePoint();
515                 //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
516                 //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
```

```
517                 //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                    ↪  constants.Itself);
518
519                 //var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
520                 //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
                    ↪  unaryOne);
521                 //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
522                 //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
523                 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
524                 //var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
525
526                 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
527
528                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache3);
529
530                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
                    ↪  linkToItsFrequencyNumberConverter);
531                 var optimalVariantConverter = new
                    ↪  OptimalVariantConverter<ulong>(scope3.Links.Unsync,
                    ↪  sequenceToItsLocalElementLevelsConverter);
532
533                 var compressed1 = new ulong[arrays.Length];
534                 var compressed2 = new ulong[arrays.Length];
535                 var compressed3 = new ulong[arrays.Length];
536
537                 var START = 0;
538                 var END = arrays.Length;
539
540                 //for (int i = START; i < END; i++)
541                 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
542
543                 var initialCount1 = scope2.Links.Unsync.Count();
544
545                 var sw1 = Stopwatch.StartNew();
546
547                 for (int i = START; i < END; i++)
548                 {
549                     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
550                     compressed1[i] = compressor1.Convert(arrays[i]);
551                 }
552
553                 var elapsed1 = sw1.Elapsed;
554
555                 var balancedVariantConverter2 = new
                    ↪  BalancedVariantConverter<ulong>(scope2.Links.Unsync);
556
557                 var initialCount2 = scope2.Links.Unsync.Count();
558
559                 var sw2 = Stopwatch.StartNew();
560
561                 for (int i = START; i < END; i++)
562                 {
563                     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
564                 }
565
566                 var elapsed2 = sw2.Elapsed;
567
568                 for (int i = START; i < END; i++)
569                 {
570                     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
571                 }
572
573                 var initialCount3 = scope3.Links.Unsync.Count();
574
575                 var sw3 = Stopwatch.StartNew();
576
577                 for (int i = START; i < END; i++)
578                 {
579                     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
580                     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
```

```
581                 }
582
583                 var elapsed3 = sw3.Elapsed;
584
585                 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
                    ↪  Optimal variant: {elapsed3}");
586
587                 // Assert.True(elapsed1 > elapsed2);
588
589                 // Checks
590                 for (int i = START; i < END; i++)
591                 {
592                     var sequence1 = compressed1[i];
593                     var sequence2 = compressed2[i];
594                     var sequence3 = compressed3[i];
595
596                     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪  scope1.Links.Unsync);
597
598                     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪  scope2.Links.Unsync);
599
600                     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
                        ↪  scope3.Links.Unsync);
601
602                     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
                        ↪  link.IsPartialPoint());
603                     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
                        ↪  link.IsPartialPoint());
604                     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
                        ↪  link.IsPartialPoint());
605
606                     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
607                     //    Assert.False(structure1 == structure2);
608                     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
609                     //    Assert.False(structure3 == structure2);
610
611                     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
612                     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
613                 }
614
615                 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
                    ↪  totalCharacters);
616                 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
                    ↪  totalCharacters);
617                 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
                    ↪  totalCharacters);
618
619                 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
                    ↪  totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
                    ↪  totalCharacters}");
620
621                 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
                    ↪  scope2.Links.Unsync.Count() - initialCount2);
622                 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
                    ↪  scope2.Links.Unsync.Count() - initialCount2);
623
624                 var duplicateProvider1 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
625                 var duplicateProvider2 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
626                 var duplicateProvider3 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
627
628                 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
629                 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
630                 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
631
632                 var duplicates1 = duplicateCounter1.Count();
633
634                 ConsoleHelpers.Debug("------");
635
636                 var duplicates2 = duplicateCounter2.Count();
637
638                 ConsoleHelpers.Debug("------");
```

```
639
640                 var duplicates3 = duplicateCounter3.Count();
641
642                 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
643
644                 linkFrequenciesCache1.ValidateFrequencies();
645                 linkFrequenciesCache3.ValidateFrequencies();
646             }
647         }
648
649         [Fact]
650         public static void CompressionStabilityTest()
651         {
652             // TODO: Fix bug (do a separate test)
653             //const ulong minNumbers = 0;
654             //const ulong maxNumbers = 1000;
655
656             const ulong minNumbers = 10000;
657             const ulong maxNumbers = 12500;
658
659             var strings = new List<string>();
660
661             for (ulong i = minNumbers; i < maxNumbers; i++)
662             {
663                 strings.Add(i.ToString());
664             }
665
666             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
667             var totalCharacters = arrays.Select(x => x.Length).Sum();
668
669             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                ↪ SequencesOptions<ulong> { UseCompression = true,
                ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
670             using (var scope2 = new TempLinksTestScope(useSequences: true))
671             {
672                 scope1.Links.UseUnicode();
673                 scope2.Links.UseUnicode();
674
675                 //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
676                 var compressor1 = scope1.Sequences;
677                 var compressor2 = scope2.Sequences;
678
679                 var compressed1 = new ulong[arrays.Length];
680                 var compressed2 = new ulong[arrays.Length];
681
682                 var sw1 = Stopwatch.StartNew();
683
684                 var START = 0;
685                 var END = arrays.Length;
686
687                 // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
688                 // Stability issue starts at 10001 or 11000
689                 //for (int i = START; i < END; i++)
690                 //{
691                 //    var first = compressor1.Compress(arrays[i]);
692                 //    var second = compressor1.Compress(arrays[i]);
693
694                 //    if (first == second)
695                 //        compressed1[i] = first;
696                 //    else
697                 //    {
698                 //        // TODO: Find a solution for this case
699                 //    }
700                 //}
701
702                 for (int i = START; i < END; i++)
703                 {
704                     var first = compressor1.Create(arrays[i].ShiftRight());
705                     var second = compressor1.Create(arrays[i].ShiftRight());
706
707                     if (first == second)
708                     {
709                         compressed1[i] = first;
710                     }
711                     else
712                     {
713                         // TODO: Find a solution for this case
714                     }
715                 }
```

```
716
717                 var elapsed1 = sw1.Elapsed;
718
719                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
720
721                 var sw2 = Stopwatch.StartNew();
722
723                 for (int i = START; i < END; i++)
724                 {
725                     var first = balancedVariantConverter.Convert(arrays[i]);
726                     var second = balancedVariantConverter.Convert(arrays[i]);
727
728                     if (first == second)
729                     {
730                         compressed2[i] = first;
731                     }
732                 }
733
734                 var elapsed2 = sw2.Elapsed;
735
736                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪ {elapsed2}");
737
738                 Assert.True(elapsed1 > elapsed2);
739
740                 // Checks
741                 for (int i = START; i < END; i++)
742                 {
743                     var sequence1 = compressed1[i];
744                     var sequence2 = compressed2[i];
745
746                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
747                     {
748                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪ scope1.Links);
749
750                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪ scope2.Links);
751
752                         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                            ↪ link.IsPartialPoint());
753                         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                            ↪ link.IsPartialPoint());
754
755                         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                            ↪ arrays[i].Length > 3)
756                         //    Assert.False(structure1 == structure2);
757
758                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
759                     }
760                 }
761
762                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
763                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
764
765                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪ totalCharacters}");
766
767                 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
768
769                 //compressor1.ValidateFrequencies();
770             }
771         }
772
773         [Fact]
774         public static void RundomNumbersCompressionQualityTest()
775         {
776             const ulong N = 500;
777
778             //const ulong minNumbers = 10000;
779             //const ulong maxNumbers = 20000;
780
781             //var strings = new List<string>();
782
783             //for (ulong i = 0; i < N; i++)
784             //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
                ↪ maxNumbers).ToString());
785
```

```
            var strings = new List<string>();

            for (ulong i = 0; i < N; i++)
            {
                strings.Add(RandomHelpers.Default.NextUInt64().ToString());
            }

            strings = strings.Distinct().ToList();

            var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
            var totalCharacters = arrays.Select(x => x.Length).Sum();

            using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
            ↪  SequencesOptions<ulong> { UseCompression = true,
            ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
            using (var scope2 = new TempLinksTestScope(useSequences: true))
            {
                scope1.Links.UseUnicode();
                scope2.Links.UseUnicode();

                var compressor1 = scope1.Sequences;
                var compressor2 = scope2.Sequences;

                var compressed1 = new ulong[arrays.Length];
                var compressed2 = new ulong[arrays.Length];

                var sw1 = Stopwatch.StartNew();

                var START = 0;
                var END = arrays.Length;

                for (int i = START; i < END; i++)
                {
                    compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
                }

                var elapsed1 = sw1.Elapsed;

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);

                var sw2 = Stopwatch.StartNew();

                for (int i = START; i < END; i++)
                {
                    compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
                }

                var elapsed2 = sw2.Elapsed;

                Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                ↪  {elapsed2}");

                Assert.True(elapsed1 > elapsed2);

                // Checks
                for (int i = START; i < END; i++)
                {
                    var sequence1 = compressed1[i];
                    var sequence2 = compressed2[i];

                    if (sequence1 != _constants.Null && sequence2 != _constants.Null)
                    {
                        var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪  scope1.Links);

                        var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪  scope2.Links);

                        Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
                    }
                }

                Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
                Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);

                Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                ↪  totalCharacters}");

```

```
                // Can be worse than balanced variant
                //Assert.True(scope1.Links.Count() <= scope2.Links.Count());

                //compressor1.ValidateFrequencies();
            }
        }

        [Fact]
        public static void AllTreeBreakDownAtSequencesCreationBugTest()
        {
            // Made out of AllPossibleConnectionsTest test.

            //const long sequenceLength = 5; //100% bug
            const long sequenceLength = 4; //100% bug
            //const long sequenceLength = 3; //100% _no_bug (ok)

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var createResults = sequences.CreateAllVariants2(sequence);

                Global.Trash = createResults;

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }

        [Fact]
        public static void AllPossibleConnectionsTest()
        {
            const long sequenceLength = 5;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var createResults = sequences.CreateAllVariants2(sequence);
                var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());

                for (var i = 0; i < 1; i++)
                {
                    var sw1 = Stopwatch.StartNew();
                    var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();

                    var sw2 = Stopwatch.StartNew();
                    var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();

                    var sw3 = Stopwatch.StartNew();
                    var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();

                    var sw4 = Stopwatch.StartNew();
                    var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();

                    Global.Trash = searchResults3;
                    Global.Trash = searchResults4; //-V3008

                    var intersection1 = createResults.Intersect(searchResults1).ToList();
                    Assert.True(intersection1.Count == createResults.Length);

                    var intersection2 = reverseResults.Intersect(searchResults1).ToList();
                    Assert.True(intersection2.Count == reverseResults.Length);

```

```
                    var intersection0 = searchResults1.Intersect(searchResults2).ToList();
                    Assert.True(intersection0.Count == searchResults2.Count);

                    var intersection3 = searchResults2.Intersect(searchResults3).ToList();
                    Assert.True(intersection3.Count == searchResults3.Count);

                    var intersection4 = searchResults3.Intersect(searchResults4).ToList();
                    Assert.True(intersection4.Count == searchResults4.Count);
                }

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }

        [Fact(Skip = "Correct implementation is pending")]
        public static void CalculateAllUsagesTest()
        {
            const long sequenceLength = 3;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var createResults = sequences.CreateAllVariants2(sequence);

                //var reverseResults =
                ↪  sequences.CreateAllVariants2(sequence.Reverse().ToArray());

                for (var i = 0; i < 1; i++)
                {
                    var linksTotalUsages1 = new ulong[links.Count() + 1];

                    sequences.CalculateAllUsages(linksTotalUsages1);

                    var linksTotalUsages2 = new ulong[links.Count() + 1];

                    sequences.CalculateAllUsages2(linksTotalUsages2);

                    var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
                    Assert.True(intersection1.Count == linksTotalUsages2.Length);
                }

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }
    }
}
```

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

```
using System;
using Xunit;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Generic;
using Platform.Data.Doublets.Memory;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class SplitMemoryGenericLinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using<byte>(links => links.TestCRUDOperations());
            Using<ushort>(links => links.TestCRUDOperations());
            Using<uint>(links => links.TestCRUDOperations());
            Using<ulong>(links => links.TestCRUDOperations());
        }
```

```

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            UsingWithExternalReferences<byte>(links => links.TestRawNumbersCRUDOperations());
            UsingWithExternalReferences<ushort>(links => links.TestRawNumbersCRUDOperations());
            UsingWithExternalReferences<uint>(links => links.TestRawNumbersCRUDOperations());
            UsingWithExternalReferences<ulong>(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(16)); // Cannot use more because current implementation of tree cuts out 5 bits from the address space.
            Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
            Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
            Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
        }

        private static void Using<TLink>(Action<ILinks<TLink>> action)
        {
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory))
            {
                action(memory);
            }
        }

        private static void UsingWithExternalReferences<TLink>(Action<ILinks<TLink>> action)
        {
            var contants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory, SplitMemoryLinks<TLink>.DefaultLinksSizeStep, contants))
            {
                action(memory);
            }
        }
    }
}
```

1.180 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs

```
using System;
using Xunit;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Specific;
using TLink = System.UInt32;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class SplitMemoryUInt32LinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            UsingWithExternalReferences(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(500));
        }

```

```
        private static void Using(Action<ILinks<TLink>> action)
        {
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory))
            {
                action(memory);
            }
        }

        private static void UsingWithExternalReferences(Action<ILinks<TLink>> action)
        {
            var contants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory,
            ↪  UInt32SplitMemoryLinks.DefaultLinksSizeStep, contants))
            {
                action(memory);
            }
        }
    }
}
```

1.181 ./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);
            }
        }
    }
}
```

1.182 ./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs

```
using System.IO;
using Platform.Disposables;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Memory.United.Specific;
using Platform.Data.Doublets.Memory.Split.Specific;
using Platform.Memory;

namespace Platform.Data.Doublets.Tests
{
    public class TempLinksTestScope : DisposableBase
    {
        public ILinks<ulong> MemoryAdapter { get; }
        public SynchronizedLinks<ulong> Links { get; }
        public Sequences.Sequences Sequences { get; }
        public string TempFilename { get; }
        public string TempTransactionLogFilename { get; }
        private readonly bool _deleteFiles;

        public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
        ↪  useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
        ↪  useLog) { }

        public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
        ↪  true, bool useSequences = false, bool useLog = false)
        {
            _deleteFiles = deleteFiles;
            TempFilename = Path.GetTempFileName();
            TempTransactionLogFilename = Path.GetTempFileName();
            //var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
            var coreMemoryAdapter = new UInt64SplitMemoryLinks(new
                FileMappedResizableDirectMemory(TempFilename), new
            ↪  FileMappedResizableDirectMemory(Path.ChangeExtension(TempFilename, "indexes")),
            ↪  UInt64SplitMemoryLinks.DefaultLinksSizeStep, new LinksConstants<ulong>(),
            ↪  Memory.IndexTreeType.Default, useLinkedList: true);
            MemoryAdapter = useLog ? (ILinks<ulong>)new
            ↪  UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
            ↪  coreMemoryAdapter;
            Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
            if (useSequences)
            {
                Sequences = new Sequences.Sequences(Links, sequencesOptions);
            }
        }

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                Links.Unsync.DisposeIfPossible();
                if (_deleteFiles)
                {
                    DeleteFiles();
                }
            }
        }

        public void DeleteFiles()
        {
            File.Delete(TempFilename);
            File.Delete(TempTransactionLogFilename);
        }
    }
}
```

1.183 ./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
    {
```

```
13          public static void TestCRUDOperations<T>(this ILinks<T> links)
14          {
15              var constants = links.Constants;
16
17              var equalityComparer = EqualityComparer<T>.Default;
18
19              var zero = default(T);
20              var one = Arithmetic.Increment(zero);
21
22              // Create Link
23              Assert.True(equalityComparer.Equals(links.Count(), zero));
24
25              var setter = new Setter<T>(constants.Null);
26              links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
27
28              Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
29
30              var linkAddress = links.Create();
31
32              var link = new Link<T>(links.GetLink(linkAddress));
33
34              Assert.True(link.Count == 3);
35              Assert.True(equalityComparer.Equals(link.Index, linkAddress));
36              Assert.True(equalityComparer.Equals(link.Source, constants.Null));
37              Assert.True(equalityComparer.Equals(link.Target, constants.Null));
38
39              Assert.True(equalityComparer.Equals(links.Count(), one));
40
41              // Get first link
42              setter = new Setter<T>(constants.Null);
43              links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
44
45              Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
46
47              // Update link to reference itself
48              links.Update(linkAddress, linkAddress, linkAddress);
49
50              link = new Link<T>(links.GetLink(linkAddress));
51
52              Assert.True(equalityComparer.Equals(link.Source, linkAddress));
53              Assert.True(equalityComparer.Equals(link.Target, linkAddress));
54
55              // Update link to reference null (prepare for delete)
56              var updated = links.Update(linkAddress, constants.Null, constants.Null);
57
58              Assert.True(equalityComparer.Equals(updated, linkAddress));
59
60              link = new Link<T>(links.GetLink(linkAddress));
61
62              Assert.True(equalityComparer.Equals(link.Source, constants.Null));
63              Assert.True(equalityComparer.Equals(link.Target, constants.Null));
64
65              // Delete link
66              links.Delete(linkAddress);
67
68              Assert.True(equalityComparer.Equals(links.Count(), zero));
69
70              setter = new Setter<T>(constants.Null);
71              links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
72
73              Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
74          }
75
76          public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
77          {
78              // Constants
79              var constants = links.Constants;
80              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;
160             var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
161             var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
162             for (var N = 1; N < maximumOperationsPerCycle; N++)
163             {
164                 var random = new System.Random(N);
165                 var created = 0UL;
166                 var deleted = 0UL;
167                 for (var i = 0; i < N; i++)
168                 {
169                     var linksCount = addressToUInt64Converter.Convert(links.Count());
170                     var createPoint = random.NextBoolean();
171                     if (linksCount >= 2 && createPoint)
```

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

1.184 ./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Xunit;
using Platform.Disposables;
using Platform.Ranges;
using Platform.Random;
using Platform.Timestamps;
using Platform.Reflection;
using Platform.Singletons;
using Platform.Scopes;
using Platform.Counters;
using Platform.Diagnostics;
using Platform.IO;
using Platform.Memory;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Memory.United.Specific;

namespace Platform.Data.Doublets.Tests
{
    public static class UInt64LinksTests
    {
        private static readonly LinksConstants<ulong> _constants =
        ↪  Default<LinksConstants<ulong>>.Instance;

        private const long Iterations = 10 * 1024;

        #region Concept

        [Fact]
        public static void MultipleCreateAndDeleteTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  UInt64UnitedMemoryLinks>>())
            {
                new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeleti
                ↪  ons(100);
            }
        }
```

```
41
42          [Fact]
43          public static void CascadeUpdateTest()
44          {
45              var itself = _constants.Itself;
46              using (var scope = new TempLinksTestScope(useLog: true))
47              {
48                  var links = scope.Links;
49
50                  var l1 = links.Create();
51                  var l2 = links.Create();
52
53                  l2 = links.Update(l2, l2, l1, l2);
54
55                  links.CreateAndUpdate(l2, itself);
56                  links.CreateAndUpdate(l2, itself);
57
58                  l2 = links.Update(l2, l1);
59
60                  links.Delete(l2);
61
62                  Global.Trash = links.Count();
63
64                  links.Unsync.DisposeIfPossible(); // Close links to access log
65
66                  Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    ↪  e.TempTransactionLogFilename);
67              }
68          }
69
70          [Fact]
71          public static void BasicTransactionLogTest()
72          {
73              using (var scope = new TempLinksTestScope(useLog: true))
74              {
75                  var links = scope.Links;
76                  var l1 = links.Create();
77                  var l2 = links.Create();
78
79                  Global.Trash = links.Update(l2, l2, l1, l2);
80
81                  links.Delete(l1);
82
83                  links.Unsync.DisposeIfPossible(); // Close links to access log
84
85                  Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    ↪  e.TempTransactionLogFilename);
86              }
87          }
88
89          [Fact]
90          public static void TransactionAutoRevertedTest()
91          {
92              // Auto Reverted (Because no commit at transaction)
93              using (var scope = new TempLinksTestScope(useLog: true))
94              {
95                  var links = scope.Links;
96                  var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
97                  using (var transaction = transactionsLayer.BeginTransaction())
98                  {
99                      var l1 = links.Create();
100                     var l2 = links.Create();
101
102                     links.Update(l2, l2, l1, l2);
103                 }
104
105                 Assert.Equal(0UL, links.Count());
106
107                 links.Unsync.DisposeIfPossible();
108
109                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
                    ↪  cope.TempTransactionLogFilename);
110                 Assert.Single(transitions);
111             }
112         }
113
114         [Fact]
115         public static void TransactionUserCodeErrorNoDataSavedTest()
116         {
```

```
117             // User Code Error (Autoreverted), no data saved
118             var itself = _constants.Itself;
119 
120             TempLinksTestScope lastScope = null;
121             try
122             {
123                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪ useLog: true))
124                 {
125                     var links = scope.Links;
126                     var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
                        ↪ atorBase<ulong>)links.Unsync).Links;
127                     using (var transaction = transactionsLayer.BeginTransaction())
128                     {
129                         var l1 = links.CreateAndUpdate(itself, itself);
130                         var l2 = links.CreateAndUpdate(itself, itself);
131 
132                         l2 = links.Update(l2, l2, l1, l2);
133 
134                         links.CreateAndUpdate(l2, itself);
135                         links.CreateAndUpdate(l2, itself);
136 
137                         //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
                            ↪ tion>(scope.TempTransactionLogFilename);
138 
139                         l2 = links.Update(l2, l1);
140 
141                         links.Delete(l2);
142 
143                         ExceptionThrower();
144 
145                         transaction.Commit();
146                     }
147 
148                     Global.Trash = links.Count();
149                 }
150             }
151             catch
152             {
153                 Assert.False(lastScope == null);
154 
155                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
                    ↪ astScope.TempTransactionLogFilename);
156 
157                 Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
                    ↪ transitions[0].After.IsNull());
158 
159                 lastScope.DeleteFiles();
160             }
161         }
162 
163         [Fact]
164         public static void TransactionUserCodeErrorSomeDataSavedTest()
165         {
166             // User Code Error (Autoreverted), some data saved
167             var itself = _constants.Itself;
168 
169             TempLinksTestScope lastScope = null;
170             try
171             {
172                 ulong l1;
173                 ulong l2;
174 
175                 using (var scope = new TempLinksTestScope(useLog: true))
176                 {
177                     var links = scope.Links;
178                     l1 = links.CreateAndUpdate(itself, itself);
179                     l2 = links.CreateAndUpdate(itself, itself);
180 
181                     l2 = links.Update(l2, l2, l1, l2);
182 
183                     links.CreateAndUpdate(l2, itself);
184                     links.CreateAndUpdate(l2, itself);
185 
186                     links.Unsync.DisposeIfPossible();
187 
188                     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
                        ↪ scope.TempTransactionLogFilename);
189                 }
190 
```

```
191                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪ useLog: true))
192                 {
193                     var links = scope.Links;
194                     var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
195                     using (var transaction = transactionsLayer.BeginTransaction())
196                     {
197                         l2 = links.Update(l2, l1);
198 
199                         links.Delete(l2);
200 
201                         ExceptionThrower();
202 
203                         transaction.Commit();
204                     }
205 
206                     Global.Trash = links.Count();
207                 }
208             }
209             catch
210             {
211                 Assert.False(lastScope == null);
212 
213                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
                    ↪ Scope.TempTransactionLogFilename);
214 
215                 lastScope.DeleteFiles();
216             }
217         }
218 
219         [Fact]
220         public static void TransactionCommit()
221         {
222             var itself = _constants.Itself;
223 
224             var tempDatabaseFilename = Path.GetTempFileName();
225             var tempTransactionLogFilename = Path.GetTempFileName();
226 
227             // Commit
228             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
229             using (var links = new UInt64Links(memoryAdapter))
230             {
231                 using (var transaction = memoryAdapter.BeginTransaction())
232                 {
233                     var l1 = links.CreateAndUpdate(itself, itself);
234                     var l2 = links.CreateAndUpdate(itself, itself);
235 
236                     Global.Trash = links.Update(l2, l2, l1, l2);
237 
238                     links.Delete(l1);
239 
240                     transaction.Commit();
241                 }
242 
243                 Global.Trash = links.Count();
244             }
245 
246             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                    ↪ sactionLogFilename);
247         }
248 
249         [Fact]
250         public static void TransactionDamage()
251         {
252             var itself = _constants.Itself;
253 
254             var tempDatabaseFilename = Path.GetTempFileName();
255             var tempTransactionLogFilename = Path.GetTempFileName();
256 
257             // Commit
258             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
259             using (var links = new UInt64Links(memoryAdapter))
260             {
261                 using (var transaction = memoryAdapter.BeginTransaction())
262                 {
263                     var l1 = links.CreateAndUpdate(itself, itself);
264                     var l2 = links.CreateAndUpdate(itself, itself);
```

```
265
266                     Global.Trash = links.Update(l2, l2, l1, l2);
267
268                     links.Delete(l1);
269
270                     transaction.Commit();
271                 }
272
273                 Global.Trash = links.Count();
274             }
275
276             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪  sactionLogFilename);
277
278             // Damage database
279
280             FileHelpers.WriteFirst(tempTransactionLogFilename, new
                ↪  UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
281
282             // Try load damaged database
283             try
284             {
285                 // TODO: Fix
286                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪  UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
287                 using (var links = new UInt64Links(memoryAdapter))
288                 {
289                     Global.Trash = links.Count();
290                 }
291             }
292             catch (NotSupportedException ex)
293             {
294                 Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
                    ↪  yet.");
295             }
296
297             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪  sactionLogFilename);
298
299             File.Delete(tempDatabaseFilename);
300             File.Delete(tempTransactionLogFilename);
301         }
302
303         [Fact]
304         public static void Bug1Test()
305         {
306             var tempDatabaseFilename = Path.GetTempFileName();
307             var tempTransactionLogFilename = Path.GetTempFileName();
308
309             var itself = _constants.Itself;
310
311             // User Code Error (Autoreverted), some data saved
312             try
313             {
314                 ulong l1;
315                 ulong l2;
316
317                 using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
318                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
                    ↪  tempTransactionLogFilename))
319                 using (var links = new UInt64Links(memoryAdapter))
320                 {
321                     l1 = links.CreateAndUpdate(itself, itself);
322                     l2 = links.CreateAndUpdate(itself, itself);
323
324                     l2 = links.Update(l2, l2, l1, l2);
325
326                     links.CreateAndUpdate(l2, itself);
327                     links.CreateAndUpdate(l2, itself);
328                 }
329
330                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                    ↪  TransactionLogFilename);
331
332                 using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
333                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
                    ↪  tempTransactionLogFilename))
334                 using (var links = new UInt64Links(memoryAdapter))
335                 {
```

```
                using (var transaction = memoryAdapter.BeginTransaction())
                {
                    l2 = links.Update(l2, l1);

                    links.Delete(l2);

                    ExceptionThrower();

                    transaction.Commit();
                }

                Global.Trash = links.Count();
            }
        }
        catch
        {
            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
            ↪  TransactionLogFilename);
        }

        File.Delete(tempDatabaseFilename);
        File.Delete(tempTransactionLogFilename);
    }

    private static void ExceptionThrower() => throw new InvalidOperationException();

    [Fact]
    public static void PathsTest()
    {
        var source = _constants.SourcePart;
        var target = _constants.TargetPart;

        using (var scope = new TempLinksTestScope())
        {
            var links = scope.Links;
            var l1 = links.CreatePoint();
            var l2 = links.CreatePoint();

            var r1 = links.GetByKeys(l1, source, target, source);
            var r2 = links.CheckPathExistance(l2, l2, l2, l2);
        }
    }

    [Fact]
    public static void RecursiveStringFormattingTest()
    {
        using (var scope = new TempLinksTestScope(useSequences: true))
        {
            var links = scope.Links;
            var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.

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

            var ab = links.GetOrCreate(a, b);
            var cb = links.GetOrCreate(c, b);
            var ac = links.GetOrCreate(a, c);

            a = links.Update(a, c, b);
            b = links.Update(b, a, c);
            c = links.Update(c, a, b);

            Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
            Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
            Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));

            Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
            ↪  "(5:(4:5 (6:5 4)) 6)");
            Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
            ↪  "(6:(5:(4:5 6) 6) 4)");
            Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
            ↪  "(4:(5:4 (6:5 4)) 6)");

            // TODO: Think how to build balanced syntax tree while formatting structure (eg.
            ↪  "(4:(5:4 6) (6:5 4)") instead of "(4:(5:4 (6:5 4)) 6)"

            Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
            ↪  "{{5}{5}{4}{6}}");
```

```
409                 Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                    ↪  "{{5}{6}{6}{4}}");
410                 Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                    ↪  "{{4}{5}{4}{6}}");
411             }
412         }
413
414         private static void DefaultFormatter(StringBuilder sb, ulong link)
415         {
416             sb.Append(link.ToString());
417         }
418
419         #endregion
420
421         #region Performance
422
423         /*
424         public static void RunAllPerformanceTests()
425         {
426             try
427             {
428                 links.TestLinksInSteps();
429             }
430             catch (Exception ex)
431             {
432                 ex.WriteToConsole();
433             }
434
435             return;
436
437             try
438             {
439                 //ThreadPool.SetMaxThreads(2, 2);
440
441                 // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    ↪  результат
442                 // Также это дополнительно помогает в отладке
443                 // Увеличивает вероятность попадания информации в кэши
444                 for (var i = 0; i < 10; i++)
445                 {
446                     //0 - 10 ГБ
447                     //Каждые 100 МБ срез цифр
448
449                     //links.TestGetSourceFunction();
450                     //links.TestGetSourceFunctionInParallel();
451                     //links.TestGetTargetFunction();
452                     //links.TestGetTargetFunctionInParallel();
453                     links.Create64BillionLinks();
454
455                     links.TestRandomSearchFixed();
456                     //links.Create64BillionLinksInParallel();
457                     links.TestEachFunction();
458                     //links.TestForeach();
459                     //links.TestParallelForeach();
460                 }
461
462                 links.TestDeletionOfAllLinks();
463
464             }
465             catch (Exception ex)
466             {
467                 ex.WriteToConsole();
468             }
469         }*/
470
471         /*
472         public static void TestLinksInSteps()
473         {
474             const long gibibyte = 1024 * 1024 * 1024;
475             const long mebibyte = 1024 * 1024;
476
477             var totalLinksToCreate = gibibyte /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478             var linksStep = 102 * mebibyte /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
479
480             var creationMeasurements = new List<TimeSpan>();
481             var searchMeasuremets = new List<TimeSpan>();
482             var deletionMeasurements = new List<TimeSpan>();
```

```
483
484             GetBaseRandomLoopOverhead(linksStep);
485             GetBaseRandomLoopOverhead(linksStep);
486
487             var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
488
489             ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
490
491             var loops = totalLinksToCreate / linksStep;
492
493             for (int i = 0; i < loops; i++)
494             {
495                 creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
496                 searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));
497
498                 Console.Write("\rC + S {0}/{1}", i + 1, loops);
499             }
500
501             ConsoleHelpers.Debug();
502
503             for (int i = 0; i < loops; i++)
504             {
505                 deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
506
507                 Console.Write("\rD {0}/{1}", i + 1, loops);
508             }
509
510             ConsoleHelpers.Debug();
511
512             ConsoleHelpers.Debug("C S D");
513
514             for (int i = 0; i < loops; i++)
515             {
516                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
    ↪  searchMeasuremets[i], deletionMeasurements[i]);
517             }
518
519             ConsoleHelpers.Debug("C S D (no overhead)");
520
521             for (int i = 0; i < loops; i++)
522             {
523                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
    ↪  searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
524             }
525
526             ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
    ↪  links.Total);
527         }
528
529         private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
    ↪  amountToCreate)
530         {
531             for (long i = 0; i < amountToCreate; i++)
532                 links.Create(0, 0);
533         }
534
535          private static TimeSpan GetBaseRandomLoopOverhead(long loops)
536          {
537              return Measure(() =>
538              {
539                  ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
540                  ulong result = 0;
541                  for (long i = 0; i < loops; i++)
542                  {
543                      var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544                      var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
545
546                      result += maxValue + source + target;
547                  }
548                  Global.Trash = result;
549              });
550          }
551           */
552
553          [Fact(Skip = "performance test")]
554          public static void GetSourceTest()
555          {
556              using (var scope = new TempLinksTestScope())
557              {
```

```
558                 var links = scope.Links;
559                 ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
                    ↪  Iterations);
560
561                 ulong counter = 0;
562
563                 //var firstLink = links.First();
564                 // Создаём одну связь, из которой будет производить считывание
565                 var firstLink = links.Create();
566
567                 var sw = Stopwatch.StartNew();
568
569                 // Тестируем саму функцию
570                 for (ulong i = 0; i < Iterations; i++)
571                 {
572                     counter += links.GetSource(firstLink);
573                 }
574
575                 var elapsedTime = sw.Elapsed;
576
577                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
578
579                 // Удаляем связь, из которой производилось считывание
580                 links.Delete(firstLink);
581
582                 ConsoleHelpers.Debug(
583                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
584                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
585             }
586         }
587
588         [Fact(Skip = "performance test")]
589         public static void GetSourceInParallel()
590         {
591             using (var scope = new TempLinksTestScope())
592             {
593                 var links = scope.Links;
594                 ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
                    ↪  parallel.", Iterations);
595
596                 long counter = 0;
597
598                 //var firstLink = links.First();
599                 var firstLink = links.Create();
600
601                 var sw = Stopwatch.StartNew();
602
603                 // Тестируем саму функцию
604                 Parallel.For(0, Iterations, x =>
605                 {
606                     Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
607                     //Interlocked.Increment(ref counter);
608                 });
609
610                 var elapsedTime = sw.Elapsed;
611
612                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
613
614                 links.Delete(firstLink);
615
616                 ConsoleHelpers.Debug(
617                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
618                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
619             }
620         }
621
622         [Fact(Skip = "performance test")]
623         public static void TestGetTarget()
624         {
625             using (var scope = new TempLinksTestScope())
626             {
627                 var links = scope.Links;
628                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
                    ↪  Iterations);
629
630                 ulong counter = 0;
631
```

```
632                 //var firstLink = links.First();
633                 var firstLink = links.Create();
634
635                 var sw = Stopwatch.StartNew();
636
637                 for (ulong i = 0; i < Iterations; i++)
638                 {
639                     counter += links.GetTarget(firstLink);
640                 }
641
642                 var elapsedTime = sw.Elapsed;
643
644                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
645
646                 links.Delete(firstLink);
647
648                 ConsoleHelpers.Debug(
649                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
650                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
651             }
652         }
653
654         [Fact(Skip = "performance test")]
655         public static void TestGetTargetInParallel()
656         {
657             using (var scope = new TempLinksTestScope())
658             {
659                 var links = scope.Links;
660                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
                    ↪  parallel.", Iterations);
661
662                 long counter = 0;
663
664                 //var firstLink = links.First();
665                 var firstLink = links.Create();
666
667                 var sw = Stopwatch.StartNew();
668
669                 Parallel.For(0, Iterations, x =>
670                 {
671                     Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
672                     //Interlocked.Increment(ref counter);
673                 });
674
675                 var elapsedTime = sw.Elapsed;
676
677                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
678
679                 links.Delete(firstLink);
680
681                 ConsoleHelpers.Debug(
682                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
683                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
684             }
685         }
686
687         // TODO: Заполнить базу данных перед тестом
688         /*
689         [Fact]
690         public void TestRandomSearchFixed()
691         {
692             var tempFilename = Path.GetTempFileName();
693
694             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
695             {
696                 long iterations = 64 * 1024 * 1024 /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
697
698                 ulong counter = 0;
699                 var maxLink = links.Total;
700
701                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
702
703                 var sw = Stopwatch.StartNew();
704
705                 for (var i = iterations; i > 0; i--)
706                 {
```

```
                    var source =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
                    var target =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);

                    counter += links.Search(source, target);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
↪ counter);
            }

            File.Delete(tempFilename);
        }*/

        [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
        public static void TestRandomSearchAll()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ulong counter = 0;

                var maxLink = links.Count();

                var iterations = links.Count();

                ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
                ↪ links.Count());

                var sw = Stopwatch.StartNew();

                for (var i = iterations; i > 0; i--)
                {
                    var linksAddressRange = new
                    ↪ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);

                    var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
                    var target = RandomHelpers.Default.NextUInt64(linksAddressRange);

                    counter += links.SearchOrDefault(source, target);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
                ↪ Iterations per second), c: {3}",
                     iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

        [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
        public static void TestEach()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;

                var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);

                ConsoleHelpers.Debug("Testing Each function.");

                var sw = Stopwatch.StartNew();

                links.Each(counter.IncrementAndReturnTrue);

                var elapsedTime = sw.Elapsed;

                var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
                ↪ links per second)",
                    counter, elapsedTime, (long)linksPerSecond);
```

```
779             }
780         }
781 
782         /*
783         [Fact]
784         public static void TestForeach()
785         {
786             var tempFilename = Path.GetTempFileName();
787 
788             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
789             {
790                 ulong counter = 0;
791 
792                 ConsoleHelpers.Debug("Testing foreach through links.");
793 
794                 var sw = Stopwatch.StartNew();
795 
796                 //foreach (var link in links)
797                 //{
798                 //    counter++;
799                 //}
800 
801                 var elapsedTime = sw.Elapsed;
802 
803                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
804 
805                 ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
    ↪  links per second)", counter, elapsedTime, (long)linksPerSecond);
806             }
807 
808             File.Delete(tempFilename);
809         }
810         */
811 
812         /*
813         [Fact]
814         public static void TestParallelForeach()
815         {
816             var tempFilename = Path.GetTempFileName();
817 
818             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
819             {
820 
821                 long counter = 0;
822 
823                 ConsoleHelpers.Debug("Testing parallel foreach through links.");
824 
825                 var sw = Stopwatch.StartNew();
826 
827                 //Parallel.ForEach((IEnumerable<ulong>)links, x =>
828                 //{
829                 //    Interlocked.Increment(ref counter);
830                 //});
831 
832                 var elapsedTime = sw.Elapsed;
833 
834                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
835 
836                 ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
    ↪  {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
837             }
838 
839             File.Delete(tempFilename);
840         }
841         */
842 
843         [Fact(Skip = "performance test")]
844         public static void Create64BillionLinks()
845         {
846             using (var scope = new TempLinksTestScope())
847             {
848                 var links = scope.Links;
849                 var linksBeforeTest = links.Count();
850 
851                 long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
852 
853                 ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
854 
```

```
855                 var elapsedTime = Performance.Measure(() =>
856                 {
857                     for (long i = 0; i < linksToCreate; i++)
858                     {
859                         links.Create();
860                     }
861                 });
862 
863                 var linksCreated = links.Count() - linksBeforeTest;
864                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
865 
866                 ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
867 
868                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    ↪ linksCreated, elapsedTime,
869                     (long)linksPerSecond);
870             }
871         }
872 
873         [Fact(Skip = "performance test")]
874         public static void Create64BillionLinksInParallel()
875         {
876             using (var scope = new TempLinksTestScope())
877             {
878                 var links = scope.Links;
879                 var linksBeforeTest = links.Count();
880 
881                 var sw = Stopwatch.StartNew();
882 
883                 long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
884 
885                 ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
886 
887                 Parallel.For(0, linksToCreate, x => links.Create());
888 
889                 var elapsedTime = sw.Elapsed;
890 
891                 var linksCreated = links.Count() - linksBeforeTest;
892                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
893 
894                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    ↪ linksCreated, elapsedTime,
895                     (long)linksPerSecond);
896             }
897         }
898 
899         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
900         public static void TestDeletionOfAllLinks()
901         {
902             using (var scope = new TempLinksTestScope())
903             {
904                 var links = scope.Links;
905                 var linksBeforeTest = links.Count();
906 
907                 ConsoleHelpers.Debug("Deleting all links");
908 
909                 var elapsedTime = Performance.Measure(links.DeleteAll);
910 
911                 var linksDeleted = linksBeforeTest - links.Count();
912                 var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
913 
914                 ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
                    ↪ linksDeleted, elapsedTime,
915                     (long)linksPerSecond);
916             }
917         }
918 
919         #endregion
920     }
921 }
```

1.185 ./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs

```
1  using Xunit;
2  using Platform.Random;
3  using Platform.Data.Doublets.Numbers.Unary;
4 
5  namespace Platform.Data.Doublets.Tests
6  {
7      public static class UnaryNumberConvertersTests
8      {
```

```
9           [Fact]
10          public static void ConvertersTest()
11          {
12              using (var scope = new TempLinksTestScope())
13              {
14                  const int N = 10;
15                  var links = scope.Links;
16                  var meaningRoot = links.CreatePoint();
17                  var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
18                  var powerOf2ToUnaryNumberConverter = new
                    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
19                  var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
20                  var random = new System.Random(0);
21                  ulong[] numbers = new ulong[N];
22                  ulong[] unaryNumbers = new ulong[N];
23                  for (int i = 0; i < N; i++)
24                  {
25                      numbers[i] = random.NextUInt64();
26                      unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
27                  }
28                  var fromUnaryNumberConverterUsingOrOperation = new
                    ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
29                  var fromUnaryNumberConverterUsingAddOperation = new
                    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
30                  for (int i = 0; i < N; i++)
31                  {
32                      Assert.Equal(numbers[i],
                        ↪  fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
33                      Assert.Equal(numbers[i],
                        ↪  fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
34                  }
35              }
36          }
37      }
38  }
```

1.186 ./csharp/Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs

```
1   using Xunit;
2   using Platform.Converters;
3   using Platform.Memory;
4   using Platform.Reflection;
5   using Platform.Scopes;
6   using Platform.Data.Numbers.Raw;
7   using Platform.Data.Doublets.Incrementers;
8   using Platform.Data.Doublets.Numbers.Unary;
9   using Platform.Data.Doublets.PropertyOperators;
10  using Platform.Data.Doublets.Sequences.Converters;
11  using Platform.Data.Doublets.Sequences.Indexes;
12  using Platform.Data.Doublets.Sequences.Walkers;
13  using Platform.Data.Doublets.Unicode;
14  using Platform.Data.Doublets.Memory.United.Generic;
15  using Platform.Data.Doublets.CriterionMatchers;
16
17  namespace Platform.Data.Doublets.Tests
18  {
19      public static class UnicodeConvertersTests
20      {
21          [Fact]
22          public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
23          {
24              using (var scope = new TempLinksTestScope())
25              {
26                  var links = scope.Links;
27                  var meaningRoot = links.CreatePoint();
28                  var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
29                  var powerOf2ToUnaryNumberConverter = new
                    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
30                  var addressToUnaryNumberConverter = new
                    ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
31                  var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
32                  TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                    ↪  addressToUnaryNumberConverter, unaryNumberToAddressConverter);
33              }
34          }
35
```

```
36          [Fact]
37          public static void CharAndRawNumberUnicodeSymbolConvertersTest()
38          {
39              using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪  UnitedMemoryLinks<ulong>>>())
40              {
41                  var links = scope.Use<ILinks<ulong>>();
42                  var meaningRoot = links.CreatePoint();
43                  var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
44                  var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
45                  TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                        ↪  addressToRawNumberConverter, rawNumberToAddressConverter);
46              }
47          }
48
49          private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
            ↪  meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
            ↪  numberToAddressConverter)
50          {
51              var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
52              var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
                    ↪  addressToNumberConverter, unicodeSymbolMarker);
53              var originalCharacter = 'H';
54              var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
55              var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
                    ↪  unicodeSymbolMarker);
56              var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
                    ↪  numberToAddressConverter, unicodeSymbolCriterionMatcher);
57              var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
58              Assert.Equal(originalCharacter, resultingCharacter);
59          }
60
61          [Fact]
62          public static void StringAndUnicodeSequenceConvertersTest()
63          {
64              using (var scope = new TempLinksTestScope())
65              {
66                  var links = scope.Links;
67
68                  var itself = links.Constants.Itself;
69
70                  var meaningRoot = links.CreatePoint();
71                  var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
72                  var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
73                  var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
74                  var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
75                  var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
76
77                  var powerOf2ToUnaryNumberConverter = new
                        ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
78                  var addressToUnaryNumberConverter = new
                        ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
79                  var charToUnicodeSymbolConverter = new
                        ↪  CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                        ↪  unicodeSymbolMarker);
80
81                  var unaryNumberToAddressConverter = new
                        ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                        ↪  powerOf2ToUnaryNumberConverter);
82                  var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
83                  var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                        ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
84                  var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                        ↪  frequencyPropertyMarker, frequencyMarker);
85                  var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                        ↪  frequencyPropertyOperator, frequencyIncrementer);
86                  var linkToItsFrequencyNumberConverter = new
                        ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                        ↪  unaryNumberToAddressConverter);
87                  var sequenceToItsLocalElementLevelsConverter = new
                        ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                        ↪  linkToItsFrequencyNumberConverter);
88                  var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                        ↪  sequenceToItsLocalElementLevelsConverter);
89
```

```
                var stringToUnicodeSequenceConverter = new
                ↪  StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
                ↪  index, optimalVariantConverter, unicodeSequenceMarker);

                var originalString = "Hello";

                var unicodeSequenceLink =
                ↪  stringToUnicodeSequenceConverter.Convert(originalString);

                var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
                ↪  unicodeSymbolMarker);
                var unicodeSymbolToCharConverter = new
                ↪  UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
                ↪  unicodeSymbolCriterionMatcher);

                var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
                ↪  unicodeSequenceMarker);

                var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
                ↪  unicodeSymbolCriterionMatcher.IsMatched);

                var unicodeSequenceToStringConverter = new
                ↪  UnicodeSequenceToStringConverter<ulong>(links,
                ↪  unicodeSequenceCriterionMatcher, sequenceWalker,
                ↪  unicodeSymbolToCharConverter);

                var resultingString =
                ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);

                Assert.Equal(originalString, resultingString);
            }
        }
    }
}
```

1.187 ./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>>());
            }
        }
    }
}
```

1.188 ./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

./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs, 236
./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs, 236
./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs, 237
./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 237
./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 240
./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 241
./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs, 242
./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs, 243
./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs, 257
./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs, 258
./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt64LinksTests.cs, 259
./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 259
./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs, 260
./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs, 263
./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 275
./csharp/Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 276
./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt32LinksTests.cs, 278
./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt64LinksTests.cs, 278
./csharp/Platform.Data.Doublets/CriterionMatchers/TargetMatcher.cs, 1
./csharp/Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs, 1
./csharp/Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs, 1
./csharp/Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs, 1
./csharp/Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs, 2
./csharp/Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs, 3
./csharp/Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs, 3
./csharp/Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs, 4
./csharp/Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs, 4
./csharp/Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs, 5
./csharp/Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs, 5
./csharp/Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs, 5
./csharp/Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs, 6
./csharp/Platform.Data.Doublets/Decorators/UInt32Links.cs, 6
./csharp/Platform.Data.Doublets/Decorators/UInt64Links.cs, 7
./csharp/Platform.Data.Doublets/Decorators/UniLinks.cs, 8
./csharp/Platform.Data.Doublets/Doublet.cs, 13
./csharp/Platform.Data.Doublets/DoubletComparer.cs, 15
./csharp/Platform.Data.Doublets/ILinks.cs, 15
./csharp/Platform.Data.Doublets/ILinksExtensions.cs, 15
./csharp/Platform.Data.Doublets/ISynchronizedLinks.cs, 28
./csharp/Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs, 28
./csharp/Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs, 28
./csharp/Platform.Data.Doublets/Link.cs, 29
./csharp/Platform.Data.Doublets/LinkExtensions.cs, 32
./csharp/Platform.Data.Doublets/LinksOperatorBase.cs, 32
./csharp/Platform.Data.Doublets/Memory/ILinksListMethods.cs, 32
./csharp/Platform.Data.Doublets/Memory/ILinksTreeMethods.cs, 33
./csharp/Platform.Data.Doublets/Memory/IndexTreeType.cs, 33
./csharp/Platform.Data.Doublets/Memory/LinksHeader.cs, 33
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 34
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSizeBalancedTreeMethodsBase.cs, 37
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 40
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesSizeBalancedTreeMethods.cs, 41
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 42
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsSizeBalancedTreeMethods.cs, 43
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 44
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSizeBalancedTreeMethodsBase.cs, 46
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesLinkedListMethods.cs, 49
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 50
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesSizeBalancedTreeMethods.cs, 51
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 52
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsSizeBalancedTreeMethods.cs, 53
./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinks.cs, 54
./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinksBase.cs, 56
./csharp/Platform.Data.Doublets/Memory/Split/Generic/UnusedLinksListMethods.cs, 67
./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.cs, 68
./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs, 68

./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 69
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSizeBalancedTreeMethodsBase.cs, 70
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 72
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesSizeBalancedTreeMethods.cs, 73
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 74
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsSizeBalancedTreeMethods.cs, 75
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 76
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSizeBalancedTreeMethodsBase.cs, 77
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesLinkedListMethods.cs, 78
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 78
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesSizeBalancedTreeMethods.cs, 79
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 80
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsSizeBalancedTreeMethods.cs, 81
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32SplitMemoryLinks.cs, 82
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32UnusedLinksListMethods.cs, 84
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 84
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSizeBalancedTreeMethodsBase.cs, 86
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 87
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesSizeBalancedTreeMethods.cs, 88
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 89
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsSizeBalancedTreeMethods.cs, 90
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 91
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSizeBalancedTreeMethodsBase.cs, 92
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesLinkedListMethods.cs, 93
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 94
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesSizeBalancedTreeMethods.cs, 94
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 95
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsSizeBalancedTreeMethods.cs, 96
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64SplitMemoryLinks.cs, 97
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64UnusedLinksListMethods.cs, 99
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksAvlBalancedTreeMethodsBase.cs, 100
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksRecursionlessSizeBalancedTreeMethodsBase.cs, 104
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSizeBalancedTreeMethodsBase.cs, 107
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesAvlBalancedTreeMethods.cs, 110
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 111
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesSizeBalancedTreeMethods.cs, 112
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.cs, 113
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 114
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsSizeBalancedTreeMethods.cs, 115
./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.cs, 116
./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.cs, 117
./csharp/Platform.Data.Doublets/Memory/United/Generic/UnusedLinksListMethods.cs, 124
./csharp/Platform.Data.Doublets/Memory/United/RawLink.cs, 125
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksRecursionlessSizeBalancedTreeMethodsBase.cs, 126
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSizeBalancedTreeMethodsBase.cs, 127
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 128
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesSizeBalancedTreeMethods.cs, 129
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 130
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsSizeBalancedTreeMethods.cs, 131
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnitedMemoryLinks.cs, 132
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnusedLinksListMethods.cs, 134
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs, 134
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksRecursionlessSizeBalancedTreeMethodsBase.cs, 136
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs, 137
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs, 138
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 139
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs, 140
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 141
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 142
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 143
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs, 144

./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.cs, 146
./csharp/Platform.Data.Doublets/Numbers/Raw/LongRawNumberSequenceToNumberConverter.cs, 146
./csharp/Platform.Data.Doublets/Numbers/Raw/NumberToLongRawNumberSequenceConverter.cs, 147
./csharp/Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 148
./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConveter.cs, 148
./csharp/Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 149
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 150
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 151
./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 152
./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 152
./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 153
./csharp/Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 154
./csharp/Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 157
./csharp/Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 157
./csharp/Platform.Data.Doublets/Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs, 159
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs, 160
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs, 160
./csharp/Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 160
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 161
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 161
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 164
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 166
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.cs, 166
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 166
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 167
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 168
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 168
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 168
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 169
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 170
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 170
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 171
./csharp/Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 171
./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 172
./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 173
./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 173
./csharp/Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 173
./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 174
./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 174
./csharp/Platform.Data.Doublets/Sequences/Sequences.cs, 201
./csharp/Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 212
./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs, 213
./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 216
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 216
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 216
./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 218
./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 219
./csharp/Platform.Data.Doublets/Stacks/Stack.cs, 219
./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs, 220
./csharp/Platform.Data.Doublets/SynchronizedLinks.cs, 220
./csharp/Platform.Data.Doublets/Time/DateTimeToLongRawNumberSequenceConverter.cs, 222
./csharp/Platform.Data.Doublets/Time/LongRawNumberSequenceToDateTimeConverter.cs, 222
./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs, 222
./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 224
./csharp/Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 230
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 230
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSymbolsListConverter.cs, 231
./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs, 231
./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 234
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 235
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs, 235