LinksPlatform's Platform.Data.Doublets Class Library

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

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

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

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

        private readonly TLink _targetToMatch;

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

```csharp
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.Decorators
8  {
9      public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
10     {
11         protected readonly LinksConstants<TLink> _constants;
12
13         public LinksConstants<TLink> Constants
14         {
15             [MethodImpl(MethodImplOptions.AggressiveInlining)]
16             get => _constants;
17         }
18
19         protected ILinks<TLink> _facade;
20
21         public ILinks<TLink> Facade
22         {
23             [MethodImpl(MethodImplOptions.AggressiveInlining)]
24             get => _facade;
25             [MethodImpl(MethodImplOptions.AggressiveInlining)]
26             set
27             {
28                 _facade = value;
29                 if (_links is LinksDecoratorBase<TLink> decorator)
30                 {
31                     decorator.Facade = value;
32                 }
33             }
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
38         {
39             _constants = links.Constants;
40             Facade = this;
41         }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public virtual TLink Count(IList<TLink> restrictions) => _links.Count(restrictions);
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
48             => _links.Each(handler, restrictions);
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public virtual TLink Create(IList<TLink> restrictions) => _links.Create(restrictions);
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
55             _links.Update(restrictions, substitution);
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         public virtual void Delete(IList<TLink> restrictions) => _links.Delete(restrictions);
59     }
60 }
```

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

```csharp
1  using System.Runtime.CompilerServices;
2  using Platform.Disposables;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5  #pragma warning disable CA1063 // Implement IDisposable Correctly
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      public abstract class LinksDisposableDecoratorBase<TLink> : LinksDecoratorBase<TLink>,
10         ILinks<TLink>, System.IDisposable
11     {
12         protected class DisposableWithMultipleCallsAllowed : Disposable
13         {
14             [MethodImpl(MethodImplOptions.AggressiveInlining)]
15             public DisposableWithMultipleCallsAllowed(Disposal disposal) : base(disposal) { }
16
17             protected override bool AllowMultipleDisposeCalls
```

```
17          {
18              [MethodImpl(MethodImplOptions.AggressiveInlining)]
19              get => true;
20          }
21      }
22
23      protected readonly DisposableWithMultipleCallsAllowed Disposable;
24
25      [MethodImpl(MethodImplOptions.AggressiveInlining)]
26      protected LinksDisposableDecoratorBase(ILinks<TLink> links) : base(links) => Disposable
        ↪  = new DisposableWithMultipleCallsAllowed(Dispose);
27
28      [MethodImpl(MethodImplOptions.AggressiveInlining)]
29      ~LinksDisposableDecoratorBase() => Disposable.Destruct();
30
31      [MethodImpl(MethodImplOptions.AggressiveInlining)]
32      public void Dispose() => Disposable.Dispose();
33
34      [MethodImpl(MethodImplOptions.AggressiveInlining)]
35      protected virtual void Dispose(bool manual, bool wasDisposed)
36      {
37          if (!wasDisposed)
38          {
39              _links.DisposeIfPossible();
40          }
41      }
42    }
43 }
```

1.6 ./csharp/Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.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.Decorators
8  {
9      // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
        ↪  be external (hybrid link's raw number).
10     public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
17         {
18             var links = _links;
19             links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
20             return links.Each(handler, restrictions);
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
25         {
26             // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
27             var links = _links;
28             links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
29             links.EnsureInnerReferenceExists(substitution, nameof(substitution));
30             return links.Update(restrictions, substitution);
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public override void Delete(IList<TLink> restrictions)
35         {
36             var link = restrictions[_constants.IndexPart];
37             var links = _links;
38             links.EnsureLinkExists(link, nameof(link));
39             links.Delete(link);
40         }
41     }
42 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
```

```csharp
#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

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <remarks>
    /// 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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
14          public override TLink Create(IList<TLink> restrictions) => _links.CreatePoint();

15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          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

```csharp
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

```csharp
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

```csharp
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 LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public LinksUsagesValidator(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              links.EnsureNoUsages(restrictions[_constants.IndexPart]);
18              return links.Update(restrictions, substitution);
19          }
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          public override void Delete(IList<TLink> restrictions)
23          {
24              var link = restrictions[_constants.IndexPart];
25              var links = _links;
26              links.EnsureNoUsages(link);
27              links.Delete(link);
28          }
29      }
30  }
```

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

```csharp
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 NonNullContentsLinkDeletionResolver<TLink> : LinksDecoratorBase<TLink>
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public NonNullContentsLinkDeletionResolver(ILinks<TLink> links) : base(links) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public override void Delete(IList<TLink> restrictions)
15          {
16              var linkIndex = restrictions[_constants.IndexPart];
17              var links = _links;
18              links.EnforceResetValues(linkIndex);
19              links.Delete(linkIndex);
20          }
21      }
22  }
```

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

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
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.Decorators
8   {
9       public class UInt32Links : LinksDisposableDecoratorBase<TLink>
10      {
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          public UInt32Links(ILinks<TLink> links) : base(links) { }
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          public override TLink Create(IList<TLink> restrictions) => _links.CreatePoint();
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
19          {
20              var constants = _constants;
21              var indexPartConstant = constants.IndexPart;
22              var sourcePartConstant = constants.SourcePart;
23              var targetPartConstant = constants.TargetPart;
```

```csharp
                    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

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <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>
```

```csharp
    {
        [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

```csharp
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>
    {
```

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

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

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

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

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

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

```
 86   ////                Func<T, bool> handler = link =>
 87   ////                {
 88   ////                    var matchedLink = Memory.GetLinkValue(link);
 89   ////                    var matchDecision = matchedHandler(matchedLink, matchedLink);
 90   ////                    return !Equals(matchDecision, Constants.Break);
 91   ////                };
 92   ////                if (!Memory.Each(handler, restriction))
 93   ////                    return Constants.Break;
 94   ////            }
 95   ////        }
 96   ////        else
 97   ////        {
 98   ////            if (substitution != null)
 99   ////            {
100   ////                transitions = new List<IList<T>>();
101   ////                Func<T, bool> handler = link =>
102   ////                {
103   ////                    var matchedLink = Memory.GetLinkValue(link);
104   ////                    transitions.Add(matchedLink);
105   ////                    return true;
106   ////                };
107   ////                if (!Memory.Each(handler, restriction))
108   ////                    return Constants.Break;
109   ////            }
110   ////            else
111   ////            {
112   ////                return Constants.Continue;
113   ////            }
114   ////        }
115   ////}
116   ////if (substitution != null)
117   ////{
118   ////    // Есть причина делать замену (запись)
119   ////    if (substitutedHandler != null)
120   ////    {
121   ////    }
122   ////    else
123   ////    {
124   ////    }
125   ////}
126   ////return Constants.Continue;
127
128   //if (restriction.IsNullOrEmpty()) // Create
129   //{
130   //    substitution[Constants.IndexPart] = Memory.AllocateLink();
131   //    Memory.SetLinkValue(substitution);
132   //}
133   //else if (substitution.IsNullOrEmpty()) // Delete
134   //{
135   //    Memory.FreeLink(restriction[Constants.IndexPart]);
136   //}
137   //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
138   //{
139   //    // No need to collect links to list
140   //    // Skip == Continue
141   //    // No need to check substituedHandler
142   //    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
      ↪ Constants.Break), restriction))
143   //        return Constants.Break;
144   //}
145   //else // Update
146   //{
147   //    //List<IList<T>> matchedLinks = null;
148   //    if (matchedHandler != null)
149   //    {
150   //        matchedLinks = new List<IList<T>>();
151   //        Func<T, bool> handler = link =>
152   //        {
153   //            var matchedLink = Memory.GetLinkValue(link);
154   //            var matchDecision = matchedHandler(matchedLink);
155   //            if (Equals(matchDecision, Constants.Break))
156   //                return false;
157   //            if (!Equals(matchDecision, Constants.Skip))
158   //                matchedLinks.Add(matchedLink);
159   //            return true;
160   //        };
161   //        if (!Memory.Each(handler, restriction))
162   //            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            {
```

```
233                 return substitutionHandler(before, after);
234             }
235             return constants.Continue;
236         }
237     else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
238     {
239         if (patternOrCondition.Count == 1)
240         {
241             var linkToDelete = patternOrCondition[0];
242             var before = _links.GetLink(linkToDelete);
243             if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                ↪ constants.Break))
244             {
245                 return constants.Break;
246             }
247             var after = Array.Empty<TLink>();
248             _links.Update(linkToDelete, constants.Null, constants.Null);
249             _links.Delete(linkToDelete);
250             if (matchHandler != null)
251             {
252                 return substitutionHandler(before, after);
253             }
254             return constants.Continue;
255         }
256         else
257         {
258             throw new NotSupportedException();
259         }
260     }
261     else // Replace / Update
262     {
263         if (patternOrCondition.Count == 1) //-V3125
264         {
265             var linkToUpdate = patternOrCondition[0];
266             var before = _links.GetLink(linkToUpdate);
267             if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                ↪ constants.Break))
268             {
269                 return constants.Break;
270             }
271             var after = (IList<TLink>)substitution.ToArray(); //-V3125
272             if (_equalityComparer.Equals(after[0], default))
273             {
274                 after[0] = linkToUpdate;
275             }
276             if (substitution.Count == 1)
277             {
278                 if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
279                 {
280                     after = _links.GetLink(substitution[0]);
281                     _links.Update(linkToUpdate, constants.Null, constants.Null);
282                     _links.Delete(linkToUpdate);
283                 }
284             }
285             else if (substitution.Count == 3)
286             {
287                 //Links.Update(after);
288             }
289             else
290             {
291                 throw new NotSupportedException();
292             }
293             if (matchHandler != null)
294             {
295                 return substitutionHandler(before, after);
296             }
297             return constants.Continue;
298         }
299         else
300         {
301             throw new NotSupportedException();
302         }
303     }
304 }
305
306 /// <remarks>
307 /// IList[IList[IList[T]]]
308 /// |       |       |     |||
```

```
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    }

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      /// <summary>
11      /// <para>.</para>
12      /// <para>.</para>
13      /// </summary>
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          /// <summary>
23          /// <para>.</para>
24          /// <para>.</para>
25          /// </summary>
26          /// <typeparam name="T">
27          /// <para>.</para>
28          /// <para>.</para>
29          /// </typeparam>
30          public readonly T Source;
31
32          /// <summary>
33          /// <para>.</para>
34          /// <para>.</para>
35          /// </summary>
36          /// <typeparam name="T">
37          /// <para>.</para>
38          /// <para>.</para>
39          /// </typeparam>
40          public readonly T Target;
41
42          /// <summary>
43          /// <para>.</para>
44          /// <para>.</para>
45          /// </summary>
46          /// <typeparam name="T">
47          /// <para>.</para>
48          /// <para>.</para>
49          /// </typeparam>
50          /// <param name="source">
51          /// <para>.</para>
52          /// <para>.</para>
```

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

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

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        /// <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);

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        /// <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;

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

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

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

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

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

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

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

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

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

```csharp
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));
```

```csharp
        /// <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
```

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

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

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

```csharp
219             }

221             #endregion

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];

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];

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];

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];

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];

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);

279             /// <summary>
280             /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
                →  (handler) для каждой подходящей связи.
```

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
            restrictions)
        {
            var arraySize = CheckedConverter<TLink,
                ulong>.Default.Convert(links.Count(restrictions));
            if (arraySize > 0)
            {
                var array = new IList<TLink>[arraySize];
                var filler = new ArrayFiller<IList<TLink>, TLink>(array,
                    links.Constants.Continue);
                links.Each(filler.AddAndReturnConstant, restrictions);
                return array;
            }
            else
            {
                return Array.Empty<IList<TLink>>();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
            restrictions)
        {
            var arraySize = CheckedConverter<TLink,
                ulong>.Default.Convert(links.Count(restrictions));
            if (arraySize > 0)
            {
                var array = new TLink[arraySize];
                var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
                links.Each(filler.AddFirstAndReturnConstant, restrictions);
                return array;
            }
            else
            {
                return Array.Empty<TLink>();
            }
        }
```

```csharp
        /// <summary>
        /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
        ↪  в хранилище связей.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Начало связи.</param>
        /// <param name="target">Конец связи.</param>
        /// <returns>Значение, определяющее существует ли связь.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
        ↪  => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
        ↪  default) > 0;

        #region Ensure
        // TODO: May be move to EnsureExtensions or make it both there and here

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
        ↪  restrictions)
        {
            for (var i = 0; i < restrictions.Count; i++)
            {
                if (!links.Exists(restrictions[i]))
                {
                    throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                    ↪  $"sequence[{i}]");
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
        ↪  reference, string argumentName)
        {
            if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
            {
                throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
        ↪  IList<TLink> restrictions, string argumentName)
        {
            for (int i = 0; i < restrictions.Count; i++)
            {
                links.EnsureInnerReferenceExists(restrictions[i], argumentName);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
        ↪  restrictions)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            var any = links.Constants.Any;
            for (var i = 0; i < restrictions.Count; i++)
            {
                if (!equalityComparer.Equals(restrictions[i], any) &&
                ↪  !links.Exists(restrictions[i]))
                {
                    throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                    ↪  $"sequence[{i}]");
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
        ↪  string argumentName)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
            {
                throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
            }
        }
```

```csharp
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
        ↪  link, string argumentName)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
            {
                throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
            }
        }

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
        ↪  TLink target)
        {
            if (links.Exists(source, target))
            {
                throw new LinkWithSameValueAlreadyExistsException();
            }
        }

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
        {
            if (links.HasUsages(link))
            {
                throw new ArgumentLinkHasDependenciesException<TLink>(link);
            }
        }

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
        ↪  addresses) => links.EnsureCreated(links.Create, addresses);

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
        ↪  addresses) => links.EnsureCreated(links.CreatePoint, addresses);

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
        ↪  params TLink[] addresses)
        {
            var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
            var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
            var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
            ↪  !links.Exists(x)));
            if (nonExistentAddresses.Count > 0)
            {
                var max = nonExistentAddresses.Max();
                max = uInt64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.⌋
                ↪  Convert(max),
                ↪  addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Max⌋
                ↪  imum)));
                var createdLinks = new List<TLink>();
                var equalityComparer = EqualityComparer<TLink>.Default;
                TLink createdLink = creator();
                while (!equalityComparer.Equals(createdLink, max))
                {
                    createdLinks.Add(createdLink);
                }
                for (var i = 0; i < createdLinks.Count; i++)
                {
                    if (!nonExistentAddresses.Contains(createdLinks[i]))
                    {
                        links.Delete(createdLinks[i]);
                    }
                }
            }
        }

        #endregion
```

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

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

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

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

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

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

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

        /// <summary>
        /// Обновляет связь с указанными началом (Source) и концом (Target)
```

```csharp
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))
```

```csharp
                {
                    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);
            }
```

```csharp
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));
```

```csharp
                        var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
                          ↪ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
                        if (!isStandalonePoint)
                        {
                            var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
                            if (totalUsages > 0)
                            {
                                var usages = ArrayPool.Allocate<TLink>(totalUsages);
                                var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
                                  ↪ links.Constants.Continue);
                                var i = 0L;
                                if (usagesAsSourceCount > 0)
                                {
                                    links.Each(usagesFiller.AddFirstAndReturnConstant,
                                      ↪ usagesAsSourceQuery);
                                    for (; i < usagesAsSourceCount; i++)
                                    {
                                        var usage = usages[i];
                                        if (!equalityComparer.Equals(usage, oldLinkIndex))
                                        {
                                            links.Update(usage, newLinkIndex, links.GetTarget(usage));
                                        }
                                    }
                                }
                                if (usagesAsTargetCount > 0)
                                {
                                    links.Each(usagesFiller.AddFirstAndReturnConstant,
                                      ↪ usagesAsTargetQuery);
                                    for (; i < usages.Length; i++)
                                    {
                                        var usage = usages[i];
                                        if (!equalityComparer.Equals(usage, oldLinkIndex))
                                        {
                                            links.Update(usage, links.GetSource(usage), newLinkIndex);
                                        }
                                    }
                                }
                                ArrayPool.Free(usages);
                            }
                        }
                    return newLinkIndex;
                }

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

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

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

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

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

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

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Incrementers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Incrementers
8  {
9      public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
          ↪  EqualityComparer<TLink>.Default;
12
13         private readonly TLink _frequencyMarker;
14         private readonly TLink _unaryOne;
15         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
          ↪  IIncrementer<TLink> unaryNumberIncrementer)
19             : base(links)
20         {
21             _frequencyMarker = frequencyMarker;
22             _unaryOne = unaryOne;
23             _unaryNumberIncrementer = unaryNumberIncrementer;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Increment(TLink frequency)
28         {
29             var links = _links;
30             if (_equalityComparer.Equals(frequency, default))
31             {
32                 return links.GetOrCreate(_unaryOne, _frequencyMarker);
33             }
34             var incrementedSource =
              ↪  _unaryNumberIncrementer.Increment(links.GetSource(frequency));
35             return links.GetOrCreate(incrementedSource, _frequencyMarker);
36         }
37     }
38  }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Incrementers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Incrementers
8  {
9      public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
          ↪  EqualityComparer<TLink>.Default;
12
13         private readonly TLink _unaryOne;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
          ↪  _unaryOne = unaryOne;
```

```csharp
        [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

```csharp
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)]
```

```csharp
 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
```

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

        #region IList

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

            #endregion
        }
    }
```

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

```csharp
using System.Runtime.CompilerServices;

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

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

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

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

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

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

```csharp
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

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

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

namespace Platform.Data.Doublets.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

```csharp
#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

```csharp
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;
```

```csharp
            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

```csharp
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);
```

```csharp
        [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>
```

```csharp
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;
```

```csharp
                    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

```csharp
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)
        {
```

```csharp
                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;
                    }
                }
```

```csharp
                    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)));
```

```
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;
184                 if (EqualToZero(link))
185                 {
186                     return @continue;
187                 }
188                 var linkBasePart = GetBasePartValue(link);
189                 var @break = Break;
190                 if (GreaterThan(linkBasePart, @base))
191                 {
192                     if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
193                     {
194                         return @break;
195                     }
196                 }
197                 else if (LessThan(linkBasePart, @base))
198                 {
199                     if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
200                     {
201                         return @break;
202                     }
203                 }
204                 else //if (linkBasePart == @base)
205                 {
206                     if (AreEqual(handler(GetLinkValues(link)), @break))
207                     {
208                         return @break;
209                     }
210                     if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
211                     {
212                         return @break;
213                     }
214                     if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
215                     {
216                         return @break;
217                     }
218                 }
219                 return @continue;
220             }
221
222             [MethodImpl(MethodImplOptions.AggressiveInlining)]
223             protected override void PrintNodeValue(TLink node, StringBuilder sb)
224             {
225                 ref var link = ref GetLinkDataPartReference(node);
226                 sb.Append(' ');
227                 sb.Append(link.Source);
228                 sb.Append('-');
229                 sb.Append('>');
230                 sb.Append(link.Target);
231             }
232         }
233     }
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
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() => GetHeaderReference().RootAsTarget;
38
39    [MethodImpl(MethodImplOptions.AggressiveInlining)]
40    protected override TLink GetBasePartValue(TLink link) =>
      ↪ GetLinkDataPartReference(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 GetLinkIndexPartReference(node);
52        link.LeftAsTarget = Zero;
53        link.RightAsTarget = Zero;
54        link.SizeAsTarget = Zero;
55    }
56    }
57 }
```

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

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.Split.Generic
6  {
7    public unsafe class ExternalLinksTargetsSizeBalancedTreeMethods<TLink> :
      ↪ ExternalLinksSizeBalancedTreeMethodsBase<TLink>
8    {
9        [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public ExternalLinksTargetsSizeBalancedTreeMethods(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).LeftAsTarget;
14
```

```csharp
        [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

```csharp
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;
```

```csharp
        protected readonly byte* Header;

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

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

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

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

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

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

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

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

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

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

        /// <summary>
```

```csharp
         /// 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);
         }
     }
 }
```

```csharp
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;
                }
```

```csharp
                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;
```

```csharp
            }
            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

```csharp
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);
            }
        }
```

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

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

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

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

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

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

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

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

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

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

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

```csharp
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);
    }
}
```

```csharp
using System.Runtime.CompilerServices;

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

```

```csharp
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|

```csharp
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>
    {
```

```csharp
        [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

```csharp
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
```

```csharp
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            }
```

```csharp
                    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

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

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

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

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

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

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

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

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

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

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

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

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

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

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
        indexMemory, long memoryReservationStep, LinksConstants<TLink> constants, bool
        useLinkedList)
    {
        _dataMemory = dataMemory;
        _indexMemory = indexMemory;
        _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
        _indexMemoryReservationStepInBytes = memoryReservationStep *
            LinkIndexPartSizeInBytes;
        _useLinkedList = useLinkedList;
        Constants = constants;
    }

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

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
        indexMemory)
    {
        // Read allocated links from header
        if (indexMemory.ReservedCapacity < LinkHeaderSizeInBytes)
        {
            indexMemory.ReservedCapacity = LinkHeaderSizeInBytes;
        }
        SetPointers(dataMemory, indexMemory);
        ref var header = ref GetHeaderReference();
        var allocatedLinks = ConvertToInt64(header.AllocatedLinks);
        // Adjust reserved capacity
        var minimumDataReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
        if (minimumDataReservedCapacity < dataMemory.UsedCapacity)
        {
            minimumDataReservedCapacity = dataMemory.UsedCapacity;
        }
        if (minimumDataReservedCapacity < _dataMemoryReservationStepInBytes)
        {
            minimumDataReservedCapacity = _dataMemoryReservationStepInBytes;
        }
        var minimumIndexReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
        if (minimumIndexReservedCapacity < indexMemory.UsedCapacity)
        {
            minimumIndexReservedCapacity = indexMemory.UsedCapacity;
        }
        if (minimumIndexReservedCapacity < _indexMemoryReservationStepInBytes)
        {
            minimumIndexReservedCapacity = _indexMemoryReservationStepInBytes;
        }
        // Check for alignment
```

```csharp
116             if (minimumDataReservedCapacity % _dataMemoryReservationStepInBytes > 0)
117             {
118                 minimumDataReservedCapacity = ((minimumDataReservedCapacity /
                    ↪ _dataMemoryReservationStepInBytes) * _dataMemoryReservationStepInBytes) +
                    ↪ _dataMemoryReservationStepInBytes;
119             }
120             if (minimumIndexReservedCapacity % _indexMemoryReservationStepInBytes > 0)
121             {
122                 minimumIndexReservedCapacity = ((minimumIndexReservedCapacity /
                    ↪ _indexMemoryReservationStepInBytes) * _indexMemoryReservationStepInBytes) +
                    ↪ _indexMemoryReservationStepInBytes;
123             }
124             if (dataMemory.ReservedCapacity != minimumDataReservedCapacity)
125             {
126                 dataMemory.ReservedCapacity = minimumDataReservedCapacity;
127             }
128             if (indexMemory.ReservedCapacity != minimumIndexReservedCapacity)
129             {
130                 indexMemory.ReservedCapacity = minimumIndexReservedCapacity;
131             }
132             SetPointers(dataMemory, indexMemory);
133             header = ref GetHeaderReference();
134             // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
135             // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
136             dataMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
                ↪ LinkDataPartSizeInBytes) + LinkDataPartSizeInBytes; // First link is read only
                ↪ zero link.
137             indexMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
                ↪ LinkIndexPartSizeInBytes) + LinkHeaderSizeInBytes;
138             // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
139             // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
140             header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
                ↪ LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
141         }
142
143         [MethodImpl(MethodImplOptions.AggressiveInlining)]
144         public virtual TLink Count(IList<TLink> restrictions)
145         {
146             // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
147             if (restrictions.Count == 0)
148             {
149                 return Total;
150             }
151             var constants = Constants;
152             var any = constants.Any;
153             var index = restrictions[constants.IndexPart];
154             if (restrictions.Count == 1)
155             {
156                 if (AreEqual(index, any))
157                 {
158                     return Total;
159                 }
160                 return Exists(index) ? GetOne() : GetZero();
161             }
162             if (restrictions.Count == 2)
163             {
164                 var value = restrictions[1];
165                 if (AreEqual(index, any))
166                 {
167                     if (AreEqual(value, any))
168                     {
169                         return Total; // Any - как отсутствие ограничения
170                     }
171                     var externalReferencesRange = constants.ExternalReferencesRange;
172                     if (externalReferencesRange.HasValue &&
                        ↪ externalReferencesRange.Value.Contains(value))
173                     {
174                         return Add(ExternalSourcesTreeMethods.CountUsages(value),
                            ↪ ExternalTargetsTreeMethods.CountUsages(value));
175                     }
176                     else
177                     {
178                         if (_useLinkedList)
179                         {
180                             return Add(InternalSourcesListMethods.CountUsages(value),
                                ↪ InternalTargetsTreeMethods.CountUsages(value));
181                         }
182                         else
```

```
183                     {
184                         return Add(InternalSourcesTreeMethods.CountUsages(value),
                            ↪   InternalTargetsTreeMethods.CountUsages(value));
185                     }
186                 }
187             }
188             else
189             {
190                 if (!Exists(index))
191                 {
192                     return GetZero();
193                 }
194                 if (AreEqual(value, any))
195                 {
196                     return GetOne();
197                 }
198                 ref var storedLinkValue = ref GetLinkDataPartReference(index);
199                 if (AreEqual(storedLinkValue.Source, value) ||
                        ↪   AreEqual(storedLinkValue.Target, value))
200                 {
201                     return GetOne();
202                 }
203                 return GetZero();
204             }
205         }
206         if (restrictions.Count == 3)
207         {
208             var externalReferencesRange = constants.ExternalReferencesRange;
209             var source = restrictions[constants.SourcePart];
210             var target = restrictions[constants.TargetPart];
211             if (AreEqual(index, any))
212             {
213                 if (AreEqual(source, any) && AreEqual(target, any))
214                 {
215                     return Total;
216                 }
217                 else if (AreEqual(source, any))
218                 {
219                     if (externalReferencesRange.HasValue &&
                        ↪   externalReferencesRange.Value.Contains(target))
220                     {
221                         return ExternalTargetsTreeMethods.CountUsages(target);
222                     }
223                     else
224                     {
225                         return InternalTargetsTreeMethods.CountUsages(target);
226                     }
227                 }
228                 else if (AreEqual(target, any))
229                 {
230                     if (externalReferencesRange.HasValue &&
                        ↪   externalReferencesRange.Value.Contains(source))
231                     {
232                         return ExternalSourcesTreeMethods.CountUsages(source);
233                     }
234                     else
235                     {
236                         if (_useLinkedList)
237                         {
238                             return InternalSourcesListMethods.CountUsages(source);
239                         }
240                         else
241                         {
242                             return InternalSourcesTreeMethods.CountUsages(source);
243                         }
244                     }
245                 }
246                 else //if(source != Any && target != Any)
247                 {
248                     // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
249                     TLink link;
250                     if (externalReferencesRange.HasValue)
251                     {
252                         if (externalReferencesRange.Value.Contains(source) &&
                            ↪   externalReferencesRange.Value.Contains(target))
253                         {
254                             link = ExternalSourcesTreeMethods.Search(source, target);
255                         }
```

```csharp
                              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();
```

```
328                    }
329                    return GetZero();
330                }
331            }
332            throw new NotSupportedException("Другие размеры и способы ограничений не
    ↪  поддерживаются.");
333        }

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

```csharp
                    var target = restrictions[constants.TargetPart];
                    if (AreEqual(index, any))
                    {
                        if (AreEqual(source, any) && AreEqual(target, any))
                        {
                            return Each(handler, Array.Empty<TLink>());
                        }
                        else if (AreEqual(source, any))
                        {
                            if (externalReferencesRange.HasValue &&
                            ↪  externalReferencesRange.Value.Contains(target))
                            {
                                return ExternalTargetsTreeMethods.EachUsage(target, handler);
                            }
                            else
                            {
                                return InternalTargetsTreeMethods.EachUsage(target, handler);
                            }
                        }
                        else if (AreEqual(target, any))
                        {
                            if (externalReferencesRange.HasValue &&
                            ↪  externalReferencesRange.Value.Contains(source))
                            {
                                return ExternalSourcesTreeMethods.EachUsage(source, handler);
                            }
                            else
                            {
                                if (_useLinkedList)
                                {
                                    return InternalSourcesListMethods.EachUsage(source, handler);
                                }
                                else
                                {
                                    return InternalSourcesTreeMethods.EachUsage(source, handler);
                                }
                            }
                        }
                        else //if(source != Any && target != Any)
                        {
                            TLink link;
                            if (externalReferencesRange.HasValue)
                            {
                                if (externalReferencesRange.Value.Contains(source) &&
                                ↪  externalReferencesRange.Value.Contains(target))
                                {
                                    link = ExternalSourcesTreeMethods.Search(source, target);
                                }
                                else if (externalReferencesRange.Value.Contains(source))
                                {
                                    link = InternalTargetsTreeMethods.Search(source, target);
                                }
                                else if (externalReferencesRange.Value.Contains(target))
                                {
                                    if (_useLinkedList)
                                    {
                                        link = ExternalSourcesTreeMethods.Search(source, target);
                                    }
                                    else
                                    {
                                        link = InternalSourcesTreeMethods.Search(source, target);
                                    }
                                }
                                else
                                {
                                    if (_useLinkedList ||
                                    ↪  GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                                    ↪  InternalTargetsTreeMethods.CountUsages(target)))
                                    {
                                        link = InternalTargetsTreeMethods.Search(source, target);
                                    }
                                    else
                                    {
                                        link = InternalSourcesTreeMethods.Search(source, target);
                                    }
                                }
                            }
                            else
```

```csharp
                        {
                            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))
            {
```

```csharp
                    if (externalReferencesRange.HasValue &&
                    ↪   externalReferencesRange.Value.Contains(source))
                    {
                        ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
                    }
                    else
                    {
                        if (_useLinkedList)
                        {
                            InternalSourcesListMethods.Detach(source, linkIndex);
                        }
                        else
                        {
                            InternalSourcesTreeMethods.Detach(ref
                            ↪   GetLinkIndexPartReference(source).RootAsSource, linkIndex);
                        }
                    }
                }
                if (!AreEqual(target, @null))
                {
                    if (externalReferencesRange.HasValue &&
                    ↪   externalReferencesRange.Value.Contains(target))
                    {
                        ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
                    }
                    else
                    {
                        InternalTargetsTreeMethods.Detach(ref
                        ↪   GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
                    }
                }
                source = link.Source = substitution[constants.SourcePart];
                target = link.Target = substitution[constants.TargetPart];
                if (!AreEqual(source, @null))
                {
                    if (externalReferencesRange.HasValue &&
                    ↪   externalReferencesRange.Value.Contains(source))
                    {
                        ExternalSourcesTreeMethods.Attach(ref rootAsSource, linkIndex);
                    }
                    else
                    {
                        if (_useLinkedList)
                        {
                            InternalSourcesListMethods.AttachAsLast(source, linkIndex);
                        }
                        else
                        {
                            InternalSourcesTreeMethods.Attach(ref
                            ↪   GetLinkIndexPartReference(source).RootAsSource, linkIndex);
                        }
                    }
                }
                if (!AreEqual(target, @null))
                {
                    if (externalReferencesRange.HasValue &&
                    ↪   externalReferencesRange.Value.Contains(target))
                    {
                        ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
                    }
                    else
                    {
                        InternalTargetsTreeMethods.Attach(ref
                        ↪   GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
                    }
                }
                return linkIndex;
        }

        /// <remarks>
        /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
        ↪   пространство
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Create(IList<TLink> restrictions)
        {
            ref var header = ref GetHeaderReference();
            var freeLink = header.FirstFreeLink;
```

```csharp
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;
```

```csharp
                ExternalSourcesTreeMethods = null;
                InternalTargetsTreeMethods = null;
                ExternalTargetsTreeMethods = null;
                UnusedLinksListMethods = null;
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        #region Disposable

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

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

        #endregion
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        protected override TLink GetNext(TLink element) =>
        ↪  GetLinkDataPartReference(element).Target;

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

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

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

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

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

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

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

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

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

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

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

        public TLink Source;
        public TLink Target;

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split
{
    public struct RawLinkIndexPart<TLink> : IEquatable<RawLinkIndexPart<TLink>>
```

```csharp
    {
        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

```csharp
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;
```

```csharp
        [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);
        }
    }
}
```

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

```csharp
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
```

```csharp
    {
        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)
```

```csharp
                {
                    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

```csharp
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)]
```

```csharp
        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;
        }
    }
}
```

1.52 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesSizeBalancedTreeMetho

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt32ExternalLinksSourcesSizeBalancedTreeMethods :
        UInt32ExternalLinksSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
            constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
            linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
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/UInt32ExternalLinksTargetsRecursionlessSizeBalan

```csharp
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;
```

```
57                    link.SizeAsTarget = Zero;
58                }
59            }
60    }
```

```
1    using System.Runtime.CompilerServices;
2    using Platform.Data.Doublets.Memory.Split.Generic;
3    using TLink = System.UInt32;
4
5    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7    namespace Platform.Data.Doublets.Memory.Split.Specific
8    {
9        public unsafe abstract class UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase :
         ↪   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
            ↪   UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
            ↪   constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            ↪   linksIndexParts, LinksHeader<TLink>* header)
17               : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
18           {
19               LinksDataParts = linksDataParts;
20               LinksIndexParts = linksIndexParts;
21               Header = header;
22           }
23
24           [MethodImpl(MethodImplOptions.AggressiveInlining)]
25           protected override TLink GetZero() => 0U;
26
27           [MethodImpl(MethodImplOptions.AggressiveInlining)]
28           protected override bool EqualToZero(TLink value) => value == 0U;
29
30           [MethodImpl(MethodImplOptions.AggressiveInlining)]
31           protected override bool AreEqual(TLink first, TLink second) => first == second;
32
33           [MethodImpl(MethodImplOptions.AggressiveInlining)]
34           protected override bool GreaterThanZero(TLink value) => value > 0U;
35
36           [MethodImpl(MethodImplOptions.AggressiveInlining)]
37           protected override bool GreaterThan(TLink first, TLink second) => first > second;
38
39           [MethodImpl(MethodImplOptions.AggressiveInlining)]
40           protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
41
42           [MethodImpl(MethodImplOptions.AggressiveInlining)]
43           protected override bool GreaterOrEqualThanZero(TLink value) => true; // value >= 0 is
            ↪   always true for ulong
44
45           [MethodImpl(MethodImplOptions.AggressiveInlining)]
46           protected override bool LessOrEqualThanZero(TLink value) => value == 0UL; // value is
            ↪   always >= 0 for ulong
47
48           [MethodImpl(MethodImplOptions.AggressiveInlining)]
49           protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
50
51           [MethodImpl(MethodImplOptions.AggressiveInlining)]
52           protected override bool LessThanZero(TLink value) => false; // value < 0 is always false
            ↪   for ulong
53
54           [MethodImpl(MethodImplOptions.AggressiveInlining)]
55           protected override bool LessThan(TLink first, TLink second) => first < second;
56
57           [MethodImpl(MethodImplOptions.AggressiveInlining)]
58           protected override TLink Increment(TLink value) => ++value;
59
60           [MethodImpl(MethodImplOptions.AggressiveInlining)]
61           protected override TLink Decrement(TLink value) => --value;
62
63           [MethodImpl(MethodImplOptions.AggressiveInlining)]
64           protected override TLink Add(TLink first, TLink second) => first + second;
65
66           [MethodImpl(MethodImplOptions.AggressiveInlining)]
67           protected override TLink Subtract(TLink first, TLink second) => first - second;
```

```csharp
        [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.

```csharp
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)]
```

```csharp
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

```csharp
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

```csharp
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)]
```

```csharp
            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

```csharp
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;
```

```csharp
        [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

```csharp
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;
```

```
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 }
```

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

```
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 UInt32InternalLinksTargetsSizeBalancedTreeMethods :
   ↪    UInt32InternalLinksSizeBalancedTreeMethodsBase
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public UInt32InternalLinksTargetsSizeBalancedTreeMethods(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)]
```

```csharp
    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

```csharp
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);
```

```csharp
                        _createExternalSourceTreeMethods = () => new
                        ↪ UInt32ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
                        ↪ _linksDataParts, _linksIndexParts, _header);
                        _createInternalTargetTreeMethods = () => new
                        ↪ UInt32InternalLinksTargetsSizeBalancedTreeMethods(Constants,
                        ↪ _linksDataParts, _linksIndexParts, _header);
                        _createExternalTargetTreeMethods = () => new
                        ↪ UInt32ExternalLinksTargetsSizeBalancedTreeMethods(Constants,
                        ↪ _linksDataParts, _linksIndexParts, _header);
                    }
                    else
                    {
                        _createInternalSourceTreeMethods = () => new
                        ↪ UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                        ↪ _linksDataParts, _linksIndexParts, _header);
                        _createExternalSourceTreeMethods = () => new
                        ↪ UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                        ↪ _linksDataParts, _linksIndexParts, _header);
                        _createInternalTargetTreeMethods = () => new
                        ↪ UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                        ↪ _linksDataParts, _linksIndexParts, _header);
                        _createExternalTargetTreeMethods = () => new
                        ↪ UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                        ↪ _linksDataParts, _linksIndexParts, _header);
                    }
                    Init(dataMemory, indexMemory);
                }

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

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

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

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

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

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

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

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
 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

```csharp
 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

```csharp
 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;
```

```csharp
        protected new readonly LinksHeader<TLink>* Header;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
84                }

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

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

7    namespace Platform.Data.Doublets.Memory.Split.Specific
8    {
9        public unsafe abstract class UInt64ExternalLinksSizeBalancedTreeMethodsBase :
             ↪  ExternalLinksSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
10       {
11           protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
12           protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
13           protected new readonly LinksHeader<TLink>* Header;

15           [MethodImpl(MethodImplOptions.AggressiveInlining)]
16           protected UInt64ExternalLinksSizeBalancedTreeMethodsBase(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           }

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

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

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

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

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

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

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

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

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

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

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

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

60           [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        protected override ulong Decrement(ulong value) => --value;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.67 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesSizeBalancedTreeMetho

```
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)]
```

```csharp
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  }
```

1.68 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsRecursionlessSizeBalan...

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

```
37        [MethodImpl(MethodImplOptions.AggressiveInlining)]
38        protected override TLink GetTreeRoot() => Header->RootAsTarget;
39
40        [MethodImpl(MethodImplOptions.AggressiveInlining)]
41        protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
42
43        [MethodImpl(MethodImplOptions.AggressiveInlining)]
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 }
```

1.69 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsSizeBalancedTreeMethods

```
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 UInt64ExternalLinksTargetsSizeBalancedTreeMethods :
       ↪  UInt64ExternalLinksSizeBalancedTreeMethodsBase
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public UInt64ExternalLinksTargetsSizeBalancedTreeMethods(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)]
```

```csharp
        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;
        }
    }
}
```

1.70 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksRecursionlessSizeBalancedTreeN

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe abstract class UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase :
        InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
    {
        protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
        protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
        protected new readonly LinksHeader<TLink>* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected
            UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
            constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            linksIndexParts, LinksHeader<TLink>* header)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
        }

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        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.

```csharp
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;
```

```csharp
        [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

```csharp
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];
```

```
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
   ↪          ref _linksIndexParts[link];
26      }
27  }
```

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

```
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 UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
   ↪        UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public
   ↪        UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(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(TLink node) => LinksIndexParts[node].RootAsSource;
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          protected override void ClearNode(TLink node)
48          {
49              ref var link = ref LinksIndexParts[node];
50              link.LeftAsSource = Zero;
51              link.RightAsSource = Zero;
52              link.SizeAsSource = Zero;
53          }
54
55          public override TLink Search(TLink source, TLink target) =>
   ↪        SearchCore(GetTreeRoot(source), target);
56      }
57  }
```

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

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

```csharp
#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

```csharp
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)]
```

```csharp
            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

```csharp
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;
```

```csharp
        [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);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64SplitMemoryLinks : SplitMemoryLinksBase<TLink>
    {
        private readonly Func<ILinksTreeMethods<TLink>> _createInternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createExternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createInternalTargetTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createExternalTargetTreeMethods;
        private LinksHeader<ulong>* _header;
        private RawLinkDataPart<ulong>* _linksDataParts;
        private RawLinkIndexPart<ulong>* _linksIndexParts;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪   indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }

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

```
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) { }

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        }

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        }

71        [MethodImpl(MethodImplOptions.AggressiveInlining)]
72        protected override void ResetPointers()
73        {
74            base.ResetPointers();
75            _linksDataParts = null;
76            _linksIndexParts = null;
77            _header = null;
78        }
79
```

```csharp
        [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

```csharp
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)]
```

```
26          protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
27      }
28  }
```

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

```csharp
1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using Platform.Numbers;
8  using static System.Runtime.CompilerServices.Unsafe;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Memory.United.Generic
13 {
14     public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> :
       ↪ SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
          ↪ UncheckedConverter<TLink, long>.Default;
17         private static readonly UncheckedConverter<TLink, int> _addressToInt32Converter =
          ↪ UncheckedConverter<TLink, int>.Default;
18         private static readonly UncheckedConverter<bool, TLink> _boolToAddressConverter =
          ↪ UncheckedConverter<bool, TLink>.Default;
19         private static readonly UncheckedConverter<TLink, bool> _addressToBoolConverter =
          ↪ UncheckedConverter<TLink, bool>.Default;
20         private static readonly UncheckedConverter<int, TLink> _int32ToAddressConverter =
          ↪ UncheckedConverter<int, TLink>.Default;
21
22         protected readonly TLink Break;
23         protected readonly TLink Continue;
24         protected readonly byte* Links;
25         protected readonly byte* Header;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
          ↪ byte* header)
29         {
30             Links = links;
31             Header = header;
32             Break = constants.Break;
33             Continue = constants.Continue;
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected abstract TLink GetTreeRoot();
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected abstract TLink GetBasePartValue(TLink link);
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
          ↪ rootSource, TLink rootTarget);
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
          ↪ rootSource, TLink rootTarget);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
          ↪ AsRef<LinksHeader<TLink>>(Header);
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
          ↪ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
          ↪ _addressToInt64Converter.Convert(link)));
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
56         {
57             ref var link = ref GetLinkReference(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         {
```

```csharp
            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
```

```csharp
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                  root = GetRightOrDefault(root);
249              }
250          }
251          return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
252      }
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                  {
```

```
287                    return Break;
288                }
289                current = GetNext(current);
290                if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
291                {
292                    break;
293                }
294            }
295        }
296        return Continue;
297    }
298
299    [MethodImpl(MethodImplOptions.AggressiveInlining)]
300    protected override void PrintNodeValue(TLink node, StringBuilder sb)
301    {
302        ref var link = ref GetLinkReference(node);
303        sb.Append(' ');
304        sb.Append(link.Source);
305        sb.Append('-');
306        sb.Append('>');
307        sb.Append(link.Target);
308    }
309  }
310 }
```

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

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

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

```
118             {
119                 root = GetLeftOrDefault(root);
120             }
121             else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
    ↪ node.Key > root.Key
122             {
123                 root = GetRightOrDefault(root);
124             }
125             else // node.Key == root.Key
126             {
127                 return root;
128             }
129         }
130         return Zero;
131     }
132
133     // TODO: Return indices range instead of references count
134     [MethodImpl(MethodImplOptions.AggressiveInlining)]
135     public TLink CountUsages(TLink link)
136     {
137         var root = GetTreeRoot();
138         var total = GetSize(root);
139         var totalRightIgnore = Zero;
140         while (!EqualToZero(root))
141         {
142             var @base = GetBasePartValue(root);
143             if (LessOrEqualThan(@base, link))
144             {
145                 root = GetRightOrDefault(root);
146             }
147             else
148             {
149                 totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
150                 root = GetLeftOrDefault(root);
151             }
152         }
153         root = GetTreeRoot();
154         var totalLeftIgnore = Zero;
155         while (!EqualToZero(root))
156         {
157             var @base = GetBasePartValue(root);
158             if (GreaterOrEqualThan(@base, link))
159             {
160                 root = GetLeftOrDefault(root);
161             }
162             else
163             {
164                 totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
165                 root = GetRightOrDefault(root);
166             }
167         }
168         return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
169     }
170
171     [MethodImpl(MethodImplOptions.AggressiveInlining)]
172     public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
    ↪ EachUsageCore(@base, GetTreeRoot(), handler);
173
174     // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
    ↪ low-level MSIL stack.
175     [MethodImpl(MethodImplOptions.AggressiveInlining)]
176     private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
177     {
178         var @continue = Continue;
179         if (EqualToZero(link))
180         {
181             return @continue;
182         }
183         var linkBasePart = GetBasePartValue(link);
184         var @break = Break;
185         if (GreaterThan(linkBasePart, @base))
186         {
187             if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
188             {
189                 return @break;
190             }
191         }
192         else if (LessThan(linkBasePart, @base))
193         {
```

```csharp
                    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

```csharp
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);
```

```csharp
        [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;
```

```csharp
                    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

```csharp
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)]
```

```csharp
        protected override void SetRight(TLink node, TLink right) =>
            GetLinkReference(node).RightAsSource = right;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

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

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

```csharp
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

```csharp
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)]
```

```csharp
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;

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

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

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

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

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

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

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

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

```csharp
        [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

```csharp
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;
```

```csharp
        [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

```csharp
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) { }
```

```csharp
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

```csharp
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;
```

```csharp
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        {
```

```csharp
            // 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();
                    }
```

```csharp
                    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))
                    {
```

```csharp
                        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) ||
```

```csharp
                    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;
            }
```

```csharp
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;
```

```csharp
            }
        }

        [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

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

```

```csharp
#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

```csharp
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;
```

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe abstract class UInt32LinksRecursionlessSizeBalancedTreeMethodsBase :
        LinksRecursionlessSizeBalancedTreeMethodsBase<uint>
    {
        protected new readonly RawLink<uint>* Links;
        protected new readonly LinksHeader<uint>* Header;

        protected UInt32LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<uint>
            constants, RawLink<uint>* links, LinksHeader<uint>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

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

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

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

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

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

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

```csharp
        [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

```csharp
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)]
```

```csharp
        protected override uint GetZero() => 0U;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.United.Specific
```

```csharp
{
    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

```csharp
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;
```

```csharp
        [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;
        }
    }
}
```

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

```csharp
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪  UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
    {
        public UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
        ↪  constants, RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links,
        ↪  header) { }

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

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

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

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

```csharp
        [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

```csharp
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)]
```

```csharp
        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

```csharp
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) { }
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
        ↪  memoryReservationStep, LinksConstants<uint> constants, IndexTreeType indexTreeType)
        ↪  : base(memory, memoryReservationStep, constants)
        {
            if (indexTreeType == IndexTreeType.SizeBalancedTree)
            {
                _createSourceTreeMethods = () => new
                ↪  UInt32LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
                _createTargetTreeMethods = () => new
                ↪  UInt32LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
            }
            else
            {
                _createSourceTreeMethods = () => new
                ↪  UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
                ↪  _header);
                _createTargetTreeMethods = () => new
                ↪  UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(Constants, _links,
                ↪  _header);
            }
            Init(memory, memoryReservationStep);
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
            protected override uint Add(uint first, uint second) => first + second;

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt32UnusedLinksListMethods : UnusedLinksListMethods<uint>
    {
        private readonly RawLink<uint>* _links;
        private readonly LinksHeader<uint>* _header;

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
        [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)]
```

```csharp
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 }
```

```csharp
 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)]
```

```csharp
        protected override ulong Subtract(ulong first, ulong second) => first - second;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
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;
```

```csharp
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

```csharp
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      {
```

```csharp
        public UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<ulong>
            constants, RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants,
            links, header) { }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
        UInt64LinksSizeBalancedTreeMethodsBase
    {
        public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
            RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
            { }

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

```
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 = 0UL;
54              link.RightAsSource = 0UL;
55              link.SizeAsSource = 0UL;
56          }
57      }
58  }
```

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

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Memory.United.Specific
6   {
7       public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
        ↪  UInt64LinksAvlBalancedTreeMethodsBase
8       {
9           public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
            ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
            ↪  { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override ref ulong GetLeftReference(ulong node) => ref
            ↪  Links[node].LeftAsTarget;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override ref ulong GetRightReference(ulong node) => ref
            ↪  Links[node].RightAsTarget;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
```

```csharp
19
20              [MethodImpl(MethodImplOptions.AggressiveInlining)]
21              protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
22
23              [MethodImpl(MethodImplOptions.AggressiveInlining)]
24              protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
   ↪   left;
25
26              [MethodImpl(MethodImplOptions.AggressiveInlining)]
27              protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
   ↪   right;
28
29              [MethodImpl(MethodImplOptions.AggressiveInlining)]
30              protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsTarget);
31
32              [MethodImpl(MethodImplOptions.AggressiveInlining)]
33              protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
   ↪   Links[node].SizeAsTarget, size);
34
35              [MethodImpl(MethodImplOptions.AggressiveInlining)]
36              protected override bool GetLeftIsChild(ulong node) =>
   ↪   GetLeftIsChildValue(Links[node].SizeAsTarget);
37
38              [MethodImpl(MethodImplOptions.AggressiveInlining)]
39              protected override void SetLeftIsChild(ulong node, bool value) =>
   ↪   SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);
40
41              [MethodImpl(MethodImplOptions.AggressiveInlining)]
42              protected override bool GetRightIsChild(ulong node) =>
   ↪   GetRightIsChildValue(Links[node].SizeAsTarget);
43
44              [MethodImpl(MethodImplOptions.AggressiveInlining)]
45              protected override void SetRightIsChild(ulong node, bool value) =>
   ↪   SetRightIsChildValue(ref Links[node].SizeAsTarget, value);
46
47              [MethodImpl(MethodImplOptions.AggressiveInlining)]
48              protected override sbyte GetBalance(ulong node) =>
   ↪   GetBalanceValue(Links[node].SizeAsTarget);
49
50              [MethodImpl(MethodImplOptions.AggressiveInlining)]
51              protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
   ↪   Links[node].SizeAsTarget, value);
52
53              [MethodImpl(MethodImplOptions.AggressiveInlining)]
54              protected override ulong GetTreeRoot() => Header->RootAsTarget;
55
56              [MethodImpl(MethodImplOptions.AggressiveInlining)]
57              protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
58
59              [MethodImpl(MethodImplOptions.AggressiveInlining)]
60              protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
   ↪   ulong secondSource, ulong secondTarget)
61                  => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
   ↪   secondSource);
62
63              [MethodImpl(MethodImplOptions.AggressiveInlining)]
64              protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
   ↪   ulong secondSource, ulong secondTarget)
65                  => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
   ↪   secondSource);
66
67              [MethodImpl(MethodImplOptions.AggressiveInlining)]
68              protected override void ClearNode(ulong node)
69              {
70                  ref var link = ref Links[node];
71                  link.LeftAsTarget = 0UL;
72                  link.RightAsTarget = 0UL;
73                  link.SizeAsTarget = 0UL;
74              }
75          }
76  }
```

1.107 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsRecursionlessSizeBalancedTr

```csharp
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   {
```

```csharp
        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

```csharp
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)
            { }
```

```csharp
            [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;
            }
        }
    }
```

1.109 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs

```csharp
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>
    {
```

```csharp
        private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
        private LinksHeader<ulong>* _header;
        private RawLink<ulong>* _links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }

        /// <summary>
        /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
        ↪   минимальным шагом расширения базы данных.
        /// </summary>
        /// <param name="address">Полный пусть к файлу базы данных.</param>
        /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
        ↪   байтах.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
        ↪   FileMappedResizableDirectMemory(address, memoryReservationStep),
        ↪   memoryReservationStep) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
        ↪   DefaultLinksSizeStep) { }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
        ↪   memoryReservationStep, LinksConstants<ulong> constants, IndexTreeType indexTreeType)
        ↪   : base(memory, memoryReservationStep, constants)
        {
            if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
            {
                _createSourceTreeMethods = () => new
                ↪   UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
                _createTargetTreeMethods = () => new
                ↪   UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
            }
            else if (indexTreeType == IndexTreeType.SizeBalancedTree)
            {
                _createSourceTreeMethods = () => new
                ↪   UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
                _createTargetTreeMethods = () => new
                ↪   UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
            }
            else
            {
                _createSourceTreeMethods = () => new
                ↪   UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
                ↪   _header);
                _createTargetTreeMethods = () => new
                ↪   UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods(Constants, _links,
                ↪   _header);
            }
            Init(memory, memoryReservationStep);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPointers(IResizableDirectMemory memory)
        {
            _header = (LinksHeader<ulong>*)memory.Pointer;
            _links = (RawLink<ulong>*)memory.Pointer;
            SourcesTreeMethods = _createSourceTreeMethods();
            TargetsTreeMethods = _createTargetTreeMethods();
            UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
        }

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

```csharp
        [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

```csharp
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/BigIntegerToRawNumberSequenceConverter.cs

```csharp
using System.Collections.Generic;
using System.Numerics;
```

```csharp
using Platform.Converters;
using Platform.Data.Doublets.Decorators;
using Platform.Numbers;
using Platform.Reflection;
using Platform.Unsafe;

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

namespace Platform.Data.Doublets.Numbers.Raw
{
    public class BigIntegerToRawNumberSequenceConverter<TLink> : LinksDecoratorBase<TLink>,
      IConverter<BigInteger, TLink>
    where TLink : struct
    {
        public readonly EqualityComparer<TLink> EqualityComparer =
          EqualityComparer<TLink>.Default;
        private readonly IConverter<TLink> _addressToNumberConverter;
        private readonly IConverter<IList<TLink>, TLink> _listToSequenceConverter;
        private static readonly TLink _maximumValue = NumericType<TLink>.MaxValue;
        private static readonly TLink _bitMask = Bit.ShiftRight(_maximumValue, 1);

        public BigIntegerToRawNumberSequenceConverter(ILinks<TLink> links, IConverter<TLink>
          addressToNumberConverter, IConverter<IList<TLink>,TLink> listToSequenceConverter) :
          base(links)
        {
            _addressToNumberConverter = addressToNumberConverter;
            _listToSequenceConverter = listToSequenceConverter;
        }

        private List<TLink> GetRawNumberParts(BigInteger bigInteger)
        {
            List<TLink> rawNumbers = new();
            for (BigInteger currentBigInt = bigInteger; currentBigInt > 0; currentBigInt >>= 63)
            {
                var bigIntBytes = currentBigInt.ToByteArray();
                var bigIntWithBitMask = Bit.And(bigIntBytes.ToStructure<TLink>(), _bitMask);
                var rawNumber = _addressToNumberConverter.Convert(bigIntWithBitMask);
                rawNumbers.Add(rawNumber);
            }
            return rawNumbers;
        }

        public TLink Convert(BigInteger bigInteger)
        {
            var rawNumbers = GetRawNumberParts(bigInteger);
            return _listToSequenceConverter.Convert(rawNumbers);
        }
    }
}
```

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

```csharp
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;
```

```
27              var externalReferencesRange = constants.ExternalReferencesRange;
28              if (externalReferencesRange.HasValue &&
    ↪   externalReferencesRange.Value.Contains(source))
29              {
30                  return
    ↪   _sourceToTargetConverter.Convert(_numberToAddressConverter.Convert(source));
31              }
32              else
33              {
34                  var pair = Links.GetLink(source);
35                  var walker = new LeftSequenceWalker<TSource>(Links, new DefaultStack<TSource>(),
    ↪   (link) => externalReferencesRange.HasValue &&
    ↪   externalReferencesRange.Value.Contains(link));
36                  TTarget result = default;
37                  foreach (var element in walker.Walk(source))
38                  {
39                      result = Bit.Or(Bit.ShiftLeft(result, _bitsPerRawNumber), Convert(element));
40                  }
41                  return result;
42              }
43          }
44      }
45  }
```

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

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Converters;
4   using Platform.Numbers;
5   using Platform.Reflection;
6   using Platform.Data.Doublets.Decorators;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets.Numbers.Raw
11  {
12      public class NumberToLongRawNumberSequenceConverter<TSource, TTarget> :
    ↪   LinksDecoratorBase<TTarget>, IConverter<TSource, TTarget>
13      {
14          private static readonly Comparer<TSource> _comparer = Comparer<TSource>.Default;
15          private static readonly TSource _maximumValue = NumericType<TSource>.MaxValue;
16          private static readonly int _bitsPerRawNumber = NumericType<TTarget>.BitsSize - 1;
17          private static readonly TSource _bitMask = Bit.ShiftRight(_maximumValue,
    ↪   NumericType<TTarget>.BitsSize + 1);
18          private static readonly TSource _maximumConvertableAddress = CheckedConverter<TTarget,
    ↪   TSource>.Default.Convert(Arithmetic.Decrement(Hybrid<TTarget>.ExternalZero));
19          private static readonly UncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
    ↪   UncheckedConverter<TSource, TTarget>.Default;
20
21          private readonly IConverter<TTarget> _addressToNumberConverter;
22
23          [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  }
```

1.114   ./csharp/Platform.Data.Doublets/Numbers/Raw/RawNumberSequenceToBigIntergerConverter.cs

```csharp
1   using System;
2   using System.Numerics;
```

```csharp
using Platform.Collections.Stacks;
using Platform.Converters;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Unsafe;

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

namespace Platform.Data.Doublets.Numbers.Raw
{
    public class RawNumberSequenceToBigIntegerConverter<TLink> : LinksDecoratorBase<TLink>,
        IConverter<TLink, BigInteger>
    where TLink : struct
    {
        private readonly IConverter<TLink, TLink> _numberToAddressConverter;
        private readonly LeftSequenceWalker<TLink> _leftSequenceWalker;

        public RawNumberSequenceToBigIntegerConverter(ILinks<TLink> links, IConverter<TLink,
            TLink> numberToAddressConverter) : base(links)
        {
            _numberToAddressConverter = numberToAddressConverter;
            _leftSequenceWalker = new(links, new DefaultStack<TLink>());
        }

        public BigInteger Convert(TLink bigInteger)
        {
            var parts = _leftSequenceWalker.Walk(bigInteger);
            using var partsEnumerator = parts.GetEnumerator();
            if (!partsEnumerator.MoveNext())
            {
                throw new Exception("Raw number sequence cannot be empty.");
            }
            var nextPart = _numberToAddressConverter.Convert(partsEnumerator.Current);
            BigInteger currentBigInt = new(nextPart.ToBytes());
            while (partsEnumerator.MoveNext())
            {
                currentBigInt <<= 63;
                nextPart = _numberToAddressConverter.Convert(partsEnumerator.Current);
                currentBigInt |= new BigInteger(nextPart.ToBytes());
            }
            return currentBigInt;
        }
    }
}
```

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

```csharp
using System.Collections.Generic;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;
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 AddressToUnaryNumberConverter<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 IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
            powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
            powerOf2ToUnaryNumberConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(TLink number)
        {
            var links = _links;
            var nullConstant = links.Constants.Null;
            var target = nullConstant;
            for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
                NumericType<TLink>.BitsSize; i++)
            {
```

```
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 }
```

1.116   ./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
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 LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
       ↪  IConverter<Doublet<TLink>, TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
14
15         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
16         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public LinkToItsFrequencyNumberConveter(
20             ILinks<TLink> links,
21             IProperty<TLink, TLink> frequencyPropertyOperator,
22             IConverter<TLink> unaryNumberToAddressConverter)
23             : base(links)
24         {
25             _frequencyPropertyOperator = frequencyPropertyOperator;
26             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public TLink Convert(Doublet<TLink> doublet)
31         {
32             var links = _links;
33             var link = links.SearchOrDefault(doublet.Source, doublet.Target);
34             if (_equalityComparer.Equals(link, default))
35             {
36                 throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
37             }
38             var frequency = _frequencyPropertyOperator.Get(link);
39             if (_equalityComparer.Equals(frequency, default))
40             {
41                 return default;
42             }
43             var frequencyNumber = links.GetSource(frequency);
44             return _unaryNumberToAddressConverter.Convert(frequencyNumber);
45         }
46     }
47 }
```

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

```
1  using System.Collections.Generic;
2  using Platform.Exceptions;
3  using Platform.Ranges;
4  using Platform.Converters;
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 PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
       ↪  IConverter<int, TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
```

```
14
15        private readonly TLink[] _unaryNumberPowersOf2;
16
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]
18        public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
19        {
20            _unaryNumberPowersOf2 = new TLink[64];
21            _unaryNumberPowersOf2[0] = one;
22        }
23
24        [MethodImpl(MethodImplOptions.AggressiveInlining)]
25        public TLink Convert(int power)
26        {
27            Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
                ↪ - 1), nameof(power));
28            if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
29            {
30                return _unaryNumberPowersOf2[power];
31            }
32            var previousPowerOf2 = Convert(power - 1);
33            var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
34            _unaryNumberPowersOf2[power] = powerOf2;
35            return powerOf2;
36        }
37    }
38 }
```

1.118 ./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             {
```

```csharp
                var result = _unaryToUInt64[source];
                TLink lastValue;
                while (!_unaryToUInt64.TryGetValue(target, out lastValue))
                {
                    source = links.GetSource(target);
                    result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
                    target = links.GetTarget(target);
                }
                result = Arithmetic<TLink>.Add(result, lastValue);
                return result;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
        ↪  links, TLink unaryOne)
        {
            var unaryToUInt64 = new Dictionary<TLink, TLink>
            {
                { 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);
    }
}
```

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

```csharp
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;
                    }
```

```
38              else
39              {
40                  powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
41                  SetBit(ref target, powerOf2Index);
42                  source = links.GetTarget(source);
43              }
44          }
45      }
46      return target;
47  }
48
49  [MethodImpl(MethodImplOptions.AggressiveInlining)]
50  private static Dictionary<TLink, int>
    ↪  CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink>
    ↪  powerOf2ToUnaryNumberConverter)
51  {
52      var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
53      for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
54      {
55          unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
56      }
57      return unaryNumberPowerOf2Indicies;
58  }
59
60  [MethodImpl(MethodImplOptions.AggressiveInlining)]
61  private static void SetBit(ref TLink target, int powerOf2Index) => target =
    ↪  Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
62  }
63 }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.PropertyOperators
8  {
9      public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>, IProperties<TLink, TLink,
    ↪  TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪  EqualityComparer<TLink>.Default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public TLink GetValue(TLink @object, TLink property)
18         {
19             var links = _links;
20             var objectProperty = links.SearchOrDefault(@object, property);
21             if (_equalityComparer.Equals(objectProperty, default))
22             {
23                 return default;
24             }
25             var constants = links.Constants;
26             var any = constants.Any;
27             var query = new Link<TLink>(any, objectProperty, any);
28             var valueLink = links.SingleOrDefault(query);
29             if (valueLink == null)
30             {
31                 return default;
32             }
33             return links.GetTarget(valueLink[constants.IndexPart]);
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public void SetValue(TLink @object, TLink property, TLink value)
38         {
39             var links = _links;
40             var objectProperty = links.GetOrCreate(@object, property);
41             links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
42             links.GetOrCreate(objectProperty, value);
43         }
44     }
45 }
```

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

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

namespace Platform.Data.Doublets.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);
            }
        }
    }
}
```

```
77   }
```

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

```csharp
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.Sequences.Converters
7  {
8      public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override TLink Convert(IList<TLink> sequence)
15         {
16             var length = sequence.Count;
17             if (length < 1)
18             {
19                 return default;
20             }
21             if (length == 1)
22             {
23                 return sequence[0];
24             }
25             // Make copy of next layer
26             if (length > 2)
27             {
28                 // TODO: Try to use stackalloc (which at the moment is not working with
                    ↪  generics) but will be possible with Sigil
29                 var halvedSequence = new TLink[(length / 2) + (length % 2)];
30                 HalveSequence(halvedSequence, sequence, length);
31                 sequence = halvedSequence;
32                 length = halvedSequence.Length;
33             }
34             // Keep creating layer after layer
35             while (length > 2)
36             {
37                 HalveSequence(sequence, sequence, length);
38                 length = (length / 2) + (length % 2);
39             }
40             return _links.GetOrCreate(sequence[0], sequence[1]);
41         }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
45         {
46             var loopedLength = length - (length % 2);
47             for (var i = 0; i < loopedLength; i += 2)
48             {
49                 destination[i / 2] = _links.GetOrCreate(source[i], source[i + 1]);
50             }
51             if (length > loopedLength)
52             {
53                 destination[length / 2] = source[length - 1];
54             }
55         }
56     }
57 }
```

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

```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections;
5  using Platform.Converters;
6  using Platform.Singletons;
7  using Platform.Numbers;
8  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Sequences.Converters
13 {
14     /// <remarks>
15     /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
          ↪  Links на этапе сжатия.
```

```csharp
16    ///     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.
17    ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
   ↪  пар, а так же разом выполнить замену.
18    /// </remarks>
19    public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
20    {
21        private static readonly LinksConstants<TLink> _constants =
   ↪  Default<LinksConstants<TLink>>.Instance;
22        private static readonly EqualityComparer<TLink> _equalityComparer =
   ↪  EqualityComparer<TLink>.Default;
23        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
24
25        private static readonly TLink _zero = default;
26        private static readonly TLink _one = Arithmetic.Increment(_zero);
27
28        private readonly IConverter<IList<TLink>, TLink> _baseConverter;
29        private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
30        private readonly TLink _minFrequencyToCompress;
31        private readonly bool _doInitialFrequenciesIncrement;
32        private Doublet<TLink> _maxDoublet;
33        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            {
```

```csharp
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                           {
```

```csharp
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];
176                     }
177                 }
178                 if (w < newLength)
179                 {
180                     copy[w] = copy[r];
181                 }
182                 oldLength = newLength;
183                 ResetMaxDoublet();
184                 UpdateMaxDoublet(copy, newLength);
185             }
186             return newLength;
187         }
188
189         [MethodImpl(MethodImplOptions.AggressiveInlining)]
190         private void ResetMaxDoublet()
191         {
192             _maxDoublet = new Doublet<TLink>();
193             _maxDoubletData = new LinkFrequency<TLink>();
194         }
195
196         [MethodImpl(MethodImplOptions.AggressiveInlining)]
197         private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
198         {
199             Doublet<TLink> doublet = default;
200             for (var i = 1; i < length; i++)
201             {
202                 doublet = new Doublet<TLink>(copy[i - 1].Element, copy[i].Element);
203                 UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
204             }
205         }
206
207         [MethodImpl(MethodImplOptions.AggressiveInlining)]
208         private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
209         {
210             var frequency = data.Frequency;
211             var maxFrequency = _maxDoubletData.Frequency;
212             //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
                ↪  (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
                ↪  compression string data (and gives collisions quickly) */ _maxDoublet.Source +
                ↪  _maxDoublet.Target)))
213             if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
214                 (_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) */
215             {
216                 _maxDoublet = doublet;
217                 _maxDoubletData = data;
218             }
219         }
220     }
221 }
```

```csharp
1   using System.Collections.Generic;
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.Sequences.Converters
8   {
9       public abstract class LinksListToSequenceConverterBase<TLink> : LinksOperatorBase<TLink>,
          ↪  IConverter<IList<TLink>, TLink>
10      {
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected LinksListToSequenceConverterBase(ILinks<TLink> links) : base(links) { }
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          public abstract TLink Convert(IList<TLink> source);
16      }
17  }
```

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Collections.Lists;
4   using Platform.Converters;
5   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
6   using Platform.Data.Doublets.Sequences.Frequencies.Counters;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets.Sequences.Converters
11  {
12      public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
13      {
14          private static readonly EqualityComparer<TLink> _equalityComparer =
              ↪  EqualityComparer<TLink>.Default;
15          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
16
17          private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
              ↪  sequenceToItsLocalElementLevelsConverter) : base(links)
21              => _sequenceToItsLocalElementLevelsConverter =
                  ↪  sequenceToItsLocalElementLevelsConverter;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          public OptimalVariantConverter(ILinks<TLink> links, LinkFrequenciesCache<TLink>
              ↪  linkFrequenciesCache)
25              : this(links, new SequenceToItsLocalElementLevelsConverter<TLink>(links, new Frequen⌋
                  ↪  ciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>(linkFrequenciesCache))) {
                  ↪  }
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          public OptimalVariantConverter(ILinks<TLink> links)
29              : this(links, new LinkFrequenciesCache<TLink>(links, new
                  ↪  TotalSequenceSymbolFrequencyCounter<TLink>(links))) { }
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public override TLink Convert(IList<TLink> sequence)
33          {
34              var length = sequence.Count;
35              if (length == 1)
36              {
37                  return sequence[0];
38              }
39              if (length == 2)
40              {
41                  return _links.GetOrCreate(sequence[0], sequence[1]);
42              }
43              sequence = sequence.ToArray();
44              var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
45              while (length > 2)
46              {
47                  var levelRepeat = 1;
48                  var currentLevel = levels[0];
49                  var previousLevel = levels[0];
50                  var skipOnce = false;
51                  var w = 0;
52                  for (var i = 1; i < length; i++)
```

```
53                      {
54                          if (_equalityComparer.Equals(currentLevel, levels[i]))
55                          {
56                              levelRepeat++;
57                              skipOnce = false;
58                              if (levelRepeat == 2)
59                              {
60                                  sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
61                                  var newLevel = i >= length - 1 ?
62                                      GetPreviousLowerThanCurrentOrCurrent(previousLevel,
                                      ↪ currentLevel) :
63                                      i < 2 ?
64                                      GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
65                                      GetGreatestNeigbourLowerThanCurrentOrCurrent(previousLevel,
                                      ↪ currentLevel, levels[i + 1]);
66                                  levels[w] = newLevel;
67                                  previousLevel = currentLevel;
68                                  w++;
69                                  levelRepeat = 0;
70                                  skipOnce = true;
71                              }
72                              else if (i == length - 1)
73                              {
74                                  sequence[w] = sequence[i];
75                                  levels[w] = levels[i];
76                                  w++;
77                              }
78                          }
79                          else
80                          {
81                              currentLevel = levels[i];
82                              levelRepeat = 1;
83                              if (skipOnce)
84                              {
85                                  skipOnce = false;
86                              }
87                              else
88                              {
89                                  sequence[w] = sequence[i - 1];
90                                  levels[w] = levels[i - 1];
91                                  previousLevel = levels[w];
92                                  w++;
93                              }
94                              if (i == length - 1)
95                              {
96                                  sequence[w] = sequence[i];
97                                  levels[w] = levels[i];
98                                  w++;
99                              }
100                         }
101                     }
102                     length = w;
103                 }
104                 return _links.GetOrCreate(sequence[0], sequence[1]);
105             }
106
107             [MethodImpl(MethodImplOptions.AggressiveInlining)]
108             private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink
            ↪ current, TLink next)
109             {
110                 return _comparer.Compare(previous, next) > 0
111                     ? _comparer.Compare(previous, current) < 0 ? previous : current
112                     : _comparer.Compare(next, current) < 0 ? next : current;
113             }
114
115             [MethodImpl(MethodImplOptions.AggressiveInlining)]
116             private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
            ↪ _comparer.Compare(next, current) < 0 ? next : current;
117
118             [MethodImpl(MethodImplOptions.AggressiveInlining)]
119             private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
            ↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
120     }
121 }
```

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Converters;
```

```csharp
#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));
    }
}
```

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

```csharp
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);
    }
}
```

1.128 ./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs

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

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

namespace Platform.Data.Doublets.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;
```

```
21          }
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          public bool IsMatched(TLink sequenceCandidate)
25              => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
26              || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
                 ↪ sequenceCandidate), _links.Constants.Null);
27      }
28  }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Stacks;
4  using Platform.Data.Doublets.Sequences.HeightProviders;
5  using Platform.Data.Sequences;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences
10 {
11     public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
          ↪ ISequenceAppender<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
              ↪ EqualityComparer<TLink>.Default;
14
15         private readonly IStack<TLink> _stack;
16         private readonly ISequenceHeightProvider<TLink> _heightProvider;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
              ↪ ISequenceHeightProvider<TLink> heightProvider)
20             : base(links)
21         {
22             _stack = stack;
23             _heightProvider = heightProvider;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Append(TLink sequence, TLink appendant)
28         {
29             var cursor = sequence;
30             var links = _links;
31             while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
32             {
33                 var source = links.GetSource(cursor);
34                 var target = links.GetTarget(cursor);
35                 if (_equalityComparer.Equals(_heightProvider.Get(source),
                      ↪ _heightProvider.Get(target)))
36                 {
37                     break;
38                 }
39                 else
40                 {
41                     _stack.Push(source);
42                     cursor = target;
43                 }
44             }
45             var left = cursor;
46             var right = appendant;
47             while (!_equalityComparer.Equals(cursor = _stack.PopOrDefault(),
                  ↪ links.Constants.Null))
48             {
49                 right = links.GetOrCreate(left, right);
50                 left = cursor;
51             }
52             return links.GetOrCreate(left, right);
53         }
54     }
55 }
```

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

```
1  using System.Collections.Generic;
2  using System.Linq;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
```

```csharp
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);
    }
}
```

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

```csharp
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)
```

```
51                  {
52                      var intermediateResult = _listComparer.Compare(left.Key, right.Key);
53                      if (intermediateResult == 0)
54                      {
55                          intermediateResult = _listComparer.Compare(left.Value, right.Value);
56                      }
57                      return intermediateResult;
58                  }
59          }
60
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
63              : base(minimumStringSegmentLength: 2)
64          {
65              _links = links;
66              _sequences = sequences;
67          }
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
71          {
72              _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)
146                 {
147                     var sequenceIndex =
    ↪   _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
148                     duplicates.Add(sequenceIndex);
149                 }
150             }
151             duplicates.Sort();
152             return duplicates;
153         }
154
155         [MethodImpl(MethodImplOptions.AggressiveInlining)]
156         private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
157         {
158             if (!(_links is ILinks<ulong> ulongLinks))
159             {
160                 return;
161             }
162             var duplicatesKey = duplicatesItem.Key;
163             var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
164             Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
165             var duplicatesList = duplicatesItem.Value;
166             for (int i = 0; i < duplicatesList.Count; i++)
167             {
168                 var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
169                 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));
170                 Console.WriteLine(formatedSequenceStructure);
171                 var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
    ↪   ulongLinks);
172                 Console.WriteLine(sequenceString);
173             }
174             Console.WriteLine();
175         }
176     }
177 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
10 {
11     /// <remarks>
12     /// Can be used to operate with many CompressingConverters (to keep global frequencies data
    ↪   between them).
13     /// TODO: Extract interface to implement frequencies storage inside Links storage
14     /// </remarks>
15     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
16     {
```

```csharp
        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))
                {
```

```csharp
                        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.");
                //    }
                //}
            }
        }
    }
}
```

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

```csharp
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}";
    }
```

```
32    }
```

1.134 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.cs

```csharp
1   using System.Runtime.CompilerServices;
2   using Platform.Converters;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
7   {
8       public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
        ↪  IConverter<Doublet<TLink>, TLink>
9       {
10          private readonly LinkFrequenciesCache<TLink> _cache;
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public
            ↪  FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
            ↪  cache) => _cache = cache;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
17      }
18  }
```

1.135 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneO...

```csharp
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 MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
        ↪  SequenceSymbolFrequencyOneOffCounter<TLink>
9       {
10          private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
            ↪  ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
14              : base(links, sequenceLink, symbol)
15              => _markedSequenceMatcher = markedSequenceMatcher;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          public override TLink Count()
19          {
20              if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
21              {
22                  return default;
23              }
24              return base.Count();
25          }
26      }
27  }
```

1.136 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCoun...

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Interfaces;
4   using Platform.Numbers;
5   using Platform.Data.Sequences;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
10  {
11      public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
12      {
13          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
14          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
15
16          protected readonly ILinks<TLink> _links;
17          protected readonly TLink _sequenceLink;
18          protected readonly TLink _symbol;
19          protected TLink _total;
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
22          public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
            ↪  TLink symbol)
23          {
24              _links = links;
25              _sequenceLink = sequenceLink;
26              _symbol = symbol;
27              _total = default;
28          }
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public virtual TLink Count()
32          {
33              if (_comparer.Compare(_total, default) > 0)
34              {
35                  return _total;
36              }
37              StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
                ↪  IsElement, VisitElement);
38              return _total;
39          }
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
            ↪  _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
            ↪  IsPartialPoint
43
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 }
```

1.137 ./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 }
```

1.138 ./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    }
```

1.139   ./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    {
8        public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
9        {
10            private readonly ILinks<TLink> _links;
11
12            [MethodImpl(MethodImplOptions.AggressiveInlining)]
13            public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
14
15            [MethodImpl(MethodImplOptions.AggressiveInlining)]
16            public TLink Count(TLink symbol) => new
   ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
17        }
18    }
```

1.140   ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOff

```
1    using System.Collections.Generic;
2    using System.Runtime.CompilerServices;
3    using Platform.Interfaces;
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.Sequences.Frequencies.Counters
9    {
10        public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
11        {
12            private static readonly EqualityComparer<TLink> _equalityComparer =
   ↪  EqualityComparer<TLink>.Default;
13            private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
14
15            protected readonly ILinks<TLink> _links;
16            protected readonly TLink _symbol;
17            protected readonly HashSet<TLink> _visits;
18            protected TLink _total;
19
20            [MethodImpl(MethodImplOptions.AggressiveInlining)]
21            public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
22            {
23                _links = links;
24                _symbol = symbol;
25                _visits = new HashSet<TLink>();
26                _total = default;
27            }
28
29            [MethodImpl(MethodImplOptions.AggressiveInlining)]
30            public TLink Count()
31            {
32                if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
33                {
34                    return _total;
35                }
36                CountCore(_symbol);
37                return _total;
38            }
39
```

```
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          private void CountCore(TLink link)
42          {
43              var any = _links.Constants.Any;
44              if (_equalityComparer.Equals(_links.Count(any, link), default))
45              {
46                  CountSequenceSymbolFrequency(link);
47              }
48              else
49              {
50                  _links.Each(EachElementHandler, any, link);
51              }
52          }
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected virtual void CountSequenceSymbolFrequency(TLink link)
56          {
57              var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
       ↪  link, _symbol);
58              _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
59          }
60
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          private TLink EachElementHandler(IList<TLink> doublet)
63          {
64              var constants = _links.Constants;
65              var doubletIndex = doublet[constants.IndexPart];
66              if (_visits.Add(doubletIndex))
67              {
68                  CountCore(doubletIndex);
69              }
70              return constants.Continue;
71          }
72      }
73  }
```

1.141 ./csharp/Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Interfaces;
4   using Platform.Converters;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences.HeightProviders
9   {
10      public class CachedSequenceHeightProvider<TLink> : ISequenceHeightProvider<TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
          ↪  EqualityComparer<TLink>.Default;
13
14          private readonly TLink _heightPropertyMarker;
15          private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
16          private readonly IConverter<TLink> _addressToUnaryNumberConverter;
17          private readonly IConverter<TLink> _unaryNumberToAddressConverter;
18          private readonly IProperties<TLink, TLink, TLink> _propertyOperator;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          public CachedSequenceHeightProvider(
22              ISequenceHeightProvider<TLink> baseHeightProvider,
23              IConverter<TLink> addressToUnaryNumberConverter,
24              IConverter<TLink> unaryNumberToAddressConverter,
25              TLink heightPropertyMarker,
26              IProperties<TLink, TLink, TLink> propertyOperator)
27          {
28              _heightPropertyMarker = heightPropertyMarker;
29              _baseHeightProvider = baseHeightProvider;
30              _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
31              _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
32              _propertyOperator = propertyOperator;
33          }
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          public TLink Get(TLink sequence)
37          {
38              TLink height;
39              var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
40              if (_equalityComparer.Equals(heightValue, default))
41              {
42                  height = _baseHeightProvider.Get(sequence);
```

```
43              heightValue = _addressToUnaryNumberConverter.Convert(height);
44              _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
45          }
46          else
47          {
48              height = _unaryNumberToAddressConverter.Convert(heightValue);
49          }
50          return height;
51      }
52   }
53 }
```

1.142   ./csharp/Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```
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.HeightProviders
8 {
9     public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
     ↪  ISequenceHeightProvider<TLink>
10    {
11        private readonly ICriterionMatcher<TLink> _elementMatcher;
12
13        [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 }
```

1.143   ./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 }
```

1.144   ./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)
```

```csharp
            {
                var indexed = true;
                var i = sequence.Count;
                while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
                ↪  { }
                for (; i >= 1; i--)
                {
                    _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
                }
                return indexed;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool IsIndexedWithIncrement(TLink source, TLink target)
            {
                var frequency = _cache.GetFrequency(source, target);
                if (frequency == null)
                {
                    return false;
                }
                var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
                if (indexed)
                {
                    _cache.IncrementFrequency(source, target);
                }
                return indexed;
            }

            [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);
            }
        }
    }
```

1.145   ./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs

```csharp
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)
```

```
27          {
28              var indexed = true;
29              var i = sequence.Count;
30              while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
↪               { }
31              for (; i >= 1; i--)
32              {
33                  Increment(_links.GetOrCreate(sequence[i - 1], sequence[i]));
34              }
35              return indexed;
36          }
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          private bool IsIndexedWithIncrement(TLink source, TLink target)
40          {
41              var link = _links.SearchOrDefault(source, target);
42              var indexed = !_equalityComparer.Equals(link, default);
43              if (indexed)
44              {
45                  Increment(link);
46              }
47              return indexed;
48          }
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          private void Increment(TLink link)
52          {
53              var previousFrequency = _frequencyPropertyOperator.Get(link);
54              var frequency = _frequencyIncrementer.Increment(previousFrequency);
55              _frequencyPropertyOperator.Set(link, frequency);
56          }
57      }
58  }
```

1.146 ./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.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.Sequences.Indexes
7   {
8       public interface ISequenceIndex<TLink>
9       {
10          /// <summary>
11          /// Индексирует последовательность глобально, и возвращает значение,
12          /// определяющие была ли запрошенная последовательность проиндексирована ранее.
13          /// </summary>
14          /// <param name="sequence">Последовательность для индексации.</param>
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          bool Add(IList<TLink> sequence);
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          bool MightContain(IList<TLink> sequence);
20      }
21  }
```

1.147 ./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.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.Sequences.Indexes
7   {
8       public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
9       {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
↪           EqualityComparer<TLink>.Default;
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public SequenceIndex(ILinks<TLink> links) : base(links) { }
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public virtual bool Add(IList<TLink> sequence)
17          {
18              var indexed = true;
19              var i = sequence.Count;
```

```
20          while (--i >= 1 && (indexed =
      ↪    !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
      ↪    default))) { }
21          for (; i >= 1; i--)
22          {
23              _links.GetOrCreate(sequence[i - 1], sequence[i]);
24          }
25          return indexed;
26      }
27
28      [MethodImpl(MethodImplOptions.AggressiveInlining)]
29      public virtual bool MightContain(IList<TLink> sequence)
30      {
31          var indexed = true;
32          var i = sequence.Count;
33          while (--i >= 1 && (indexed =
      ↪    !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
      ↪    default))) { }
34          return indexed;
35      }
36  }
37 }
```

1.148 ./csharp/Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.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.Sequences.Indexes
7 {
8     public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
      ↪    EqualityComparer<TLink>.Default;
11
12         private readonly ISynchronizedLinks<TLink> _links;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public bool Add(IList<TLink> sequence)
19         {
20             var indexed = true;
21             var i = sequence.Count;
22             var links = _links.Unsync;
23             _links.SyncRoot.ExecuteReadOperation(() =>
24             {
25                 while (--i >= 1 && (indexed =
      ↪        !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
      ↪        sequence[i]), default))) { }
26             });
27             if (!indexed)
28             {
29                 _links.SyncRoot.ExecuteWriteOperation(() =>
30                 {
31                     for (; i >= 1; i--)
32                     {
33                         links.GetOrCreate(sequence[i - 1], sequence[i]);
34                     }
35                 });
36             }
37             return indexed;
38         }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public bool MightContain(IList<TLink> sequence)
42         {
43             var links = _links.Unsync;
44             return _links.SyncRoot.ExecuteReadOperation(() =>
45             {
46                 var indexed = true;
47                 var i = sequence.Count;
48                 while (--i >= 1 && (indexed =
      ↪        !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
      ↪        sequence[i]), default))) { }
49                 return indexed;
50             });
```

```
51          }
52        }
53    }
```

1.149 ./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.cs

```csharp
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.Sequences.Indexes
7    {
8        public class Unindex<TLink> : ISequenceIndex<TLink>
9        {
10            [MethodImpl(MethodImplOptions.AggressiveInlining)]
11            public virtual bool Add(IList<TLink> sequence) => false;
12
13            [MethodImpl(MethodImplOptions.AggressiveInlining)]
14            public virtual bool MightContain(IList<TLink> sequence) => true;
15        }
16    }
```

1.150 ./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs

```csharp
1    using System;
2    using System.Collections.Generic;
3    using System.Runtime.CompilerServices;
4    using System.Linq;
5    using System.Text;
6    using Platform.Collections;
7    using Platform.Collections.Sets;
8    using Platform.Collections.Stacks;
9    using Platform.Data.Exceptions;
10   using Platform.Data.Sequences;
11   using Platform.Data.Doublets.Sequences.Frequencies.Counters;
12   using Platform.Data.Doublets.Sequences.Walkers;
13   using LinkIndex = System.UInt64;
14   using Stack = System.Collections.Generic.Stack<ulong>;
15
16   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
17
18   namespace Platform.Data.Doublets.Sequences
19   {
20       partial class Sequences
21       {
22           #region Create All Variants (Not Practical)
23
24           /// <remarks>
25           /// Number of links that is needed to generate all variants for
26           /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
27           /// </remarks>
28           [MethodImpl(MethodImplOptions.AggressiveInlining)]
29           public ulong[] CreateAllVariants2(ulong[] sequence)
30           {
31               return _sync.ExecuteWriteOperation(() =>
32               {
33                   if (sequence.IsNullOrEmpty())
34                   {
35                       return Array.Empty<ulong>();
36                   }
37                   Links.EnsureLinkExists(sequence);
38                   if (sequence.Length == 1)
39                   {
40                       return sequence;
41                   }
42                   return CreateAllVariants2Core(sequence, 0, (ulong)sequence.Length - 1);
43               });
44           }
45
46           [MethodImpl(MethodImplOptions.AggressiveInlining)]
47           private ulong[] CreateAllVariants2Core(ulong[] sequence, ulong startAt, ulong stopAt)
48           {
49               if ((stopAt - startAt) == 0)
50               {
51                   return new[] { sequence[startAt] };
52               }
53               if ((stopAt - startAt) == 1)
54               {
55                   return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
56               }
57               var variants = new ulong[Platform.Numbers.Math.Catalan(stopAt - startAt)];
```

```csharp
                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((ulong)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;
```

```csharp
        }

        #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)]
        private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
        {
            if (sequence.Length == 2)
            {
                Links.Unsync.Each(sequence[0], sequence[1], handler);
            }
            else
            {
                var innerSequenceLength = sequence.Length - 1;
                for (var li = 0; li < innerSequenceLength; li++)
                {
                    var left = sequence[li];
                    var right = sequence[li + 1];
                    if (left == 0 && right == 0)
                    {
                        continue;
                    }
                    var linkIndex = li;
                    ulong[] innerSequence = null;
                    Links.Unsync.Each(doublet =>
                    {
                        if (innerSequence == null)
                        {
                            innerSequence = new ulong[innerSequenceLength];
                            for (var isi = 0; isi < linkIndex; isi++)
                            {
                                innerSequence[isi] = sequence[isi];
                            }
                            for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
                            {
                                innerSequence[isi] = sequence[isi + 1];
                            }
                        }
                        innerSequence[linkIndex] = doublet[Constants.IndexPart];
                        Each1(handler, innerSequence);
                        return Constants.Continue;
                    }, Constants.Any, left, right);
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> EachPart(params ulong[] sequence)
        {
            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
            EachPartCore(link =>
            {
                var linkIndex = link[Constants.IndexPart];
                if (!visitedLinks.Contains(linkIndex))
                {
                    visitedLinks.Add(linkIndex); // изучить почему случаются повторы
                }
                return Constants.Continue;
            }, sequence);
            return visitedLinks;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
```

```
211         {
212             var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
213             EachPartCore(link =>
214             {
215                 var linkIndex = link[Constants.IndexPart];
216                 if (!visitedLinks.Contains(linkIndex))
217                 {
218                     visitedLinks.Add(linkIndex); // изучить почему случаются повторы
219                     return handler(new LinkAddress<LinkIndex>(linkIndex));
220                 }
221                 return Constants.Continue;
222             }, sequence);
223         }
224
225         [MethodImpl(MethodImplOptions.AggressiveInlining)]
226         private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
    ↪   sequence)
227         {
228             if (sequence.IsNullOrEmpty())
229             {
230                 return;
231             }
232             Links.EnsureLinkIsAnyOrExists(sequence);
233             if (sequence.Length == 1)
234             {
235                 var link = sequence[0];
236                 if (link > 0)
237                 {
238                     handler(new LinkAddress<LinkIndex>(link));
239                 }
240                 else
241                 {
242                     Links.Each(Constants.Any, Constants.Any, handler);
243                 }
244             }
245             else if (sequence.Length == 2)
246             {
247                 //_links.Each(sequence[0], sequence[1], handler);
248                 //  o_|        x_o ...
249                 // x_|          |___|
250                 Links.Each(sequence[1], Constants.Any, doublet =>
251                 {
252                     var match = Links.SearchOrDefault(sequence[0], doublet);
253                     if (match != Constants.Null)
254                     {
255                         handler(new LinkAddress<LinkIndex>(match));
256                     }
257                     return true;
258                 });
259                 // |_x        ... x_o
260                 // |_o          |___|
261                 Links.Each(Constants.Any, sequence[0], doublet =>
262                 {
263                     var match = Links.SearchOrDefault(doublet, sequence[1]);
264                     if (match != 0)
265                     {
266                         handler(new LinkAddress<LinkIndex>(match));
267                     }
268                     return true;
269                 });
270                 //           ._x o_.
271                 //            |___|
272                 PartialStepRight(x => handler(x), sequence[0], sequence[1]);
273             }
274             else
275             {
276                 throw new NotImplementedException();
277             }
278         }
279
280         [MethodImpl(MethodImplOptions.AggressiveInlining)]
281         private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
282         {
283             Links.Unsync.Each(Constants.Any, left, doublet =>
284             {
285                 StepRight(handler, doublet, right);
286                 if (left != doublet)
287                 {
```

```csharp
                        PartialStepRight(handler, doublet, right);
                    }
                    return true;
                });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(left, Constants.Any, rightStep =>
            {
                TryStepRightUp(handler, right, rightStep);
                return true;
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
        ↪    stepFrom)
        {
            var upStep = stepFrom;
            var firstSource = Links.Unsync.GetTarget(upStep);
            while (firstSource != right && firstSource != upStep)
            {
                upStep = firstSource;
                firstSource = Links.Unsync.GetSource(upStep);
            }
            if (firstSource == right)
            {
                handler(new LinkAddress<LinkIndex>(stepFrom));
            }
        }

        // TODO: Test
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(right, Constants.Any, doublet =>
            {
                StepLeft(handler, left, doublet);
                if (right != doublet)
                {
                    PartialStepLeft(handler, left, doublet);
                }
                return true;
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(Constants.Any, right, leftStep =>
            {
                TryStepLeftUp(handler, left, leftStep);
                return true;
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
        {
            var upStep = stepFrom;
            var firstTarget = Links.Unsync.GetSource(upStep);
            while (firstTarget != left && firstTarget != upStep)
            {
                upStep = firstTarget;
                firstTarget = Links.Unsync.GetTarget(upStep);
            }
            if (firstTarget == left)
            {
                handler(new LinkAddress<LinkIndex>(stepFrom));
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool StartsWith(ulong sequence, ulong link)
        {
            var upStep = sequence;
            var firstSource = Links.Unsync.GetSource(upStep);
```

```csharp
                    while (firstSource != link && firstSource != upStep)
                    {
                        upStep = firstSource;
                        firstSource = Links.Unsync.GetSource(upStep);
                    }
                    return firstSource == link;
                }

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                private bool EndsWith(ulong sequence, ulong link)
                {
                    var upStep = sequence;
                    var lastTarget = Links.Unsync.GetTarget(upStep);
                    while (lastTarget != link && lastTarget != upStep)
                    {
                        upStep = lastTarget;
                        lastTarget = Links.Unsync.GetTarget(upStep);
                    }
                    return lastTarget == link;
                }

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
                {
                    return _sync.ExecuteReadOperation(() =>
                    {
                        var results = new List<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 linksInSequence = new HashSet<ulong>(sequence);
                            void handler(IList<LinkIndex> result)
                            {
                                var resultIndex = result[Links.Constants.IndexPart];
                                var filterPosition = 0;
                                StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
                                ↪ Links.Unsync.GetTarget,
                                    x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                                ↪ x =>
                                {
                                    if (filterPosition == sequence.Length)
                                    {
                                        filterPosition = -2; // Длиннее чем нужно
                                        return false;
                                    }
                                    if (x != sequence[filterPosition])
                                    {
                                        filterPosition = -1;
                                        return false; // Начинается иначе
                                    }
                                    filterPosition++;

                                    return true;
                                });
                                if (filterPosition == sequence.Length)
                                {
                                    results.Add(resultIndex);
                                }
                            }
                            if (sequence.Length >= 2)
                            {
                                StepRight(handler, sequence[0], sequence[1]);
                            }
                            var last = sequence.Length - 2;
```

```csharp
                    for (var i = 1; i < last; i++)
                    {
                        PartialStepRight(handler, sequence[i], sequence[i + 1]);
                    }
                    if (sequence.Length >= 3)
                    {
                        StepLeft(handler, sequence[sequence.Length - 2],
                        ↪   sequence[sequence.Length - 1]);
                    }
                }
                return results;
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                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>();
```

```csharp
512                    var sb = new StringBuilder();
513                    sb.Append('{');
514                    if (links.Exists(sequenceLink))
515                    {
516                        StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
517                            x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
                            ↪  entered.AddAndReturnVoid, x => { }, entered.DoNotContains
518                        {
519                            if (insertComma && sb.Length > 1)
520                            {
521                                sb.Append(',');
522                            }
523                            //if (entered.Contains(element))
524                            //{
525                            //    sb.Append('{');
526                            //    elementToString(sb, element);
527                            //    sb.Append('}');
528                            //}
529                            //else
530                            elementToString(sb, element);
531                            if (sb.Length < MaxSequenceFormatSize)
532                            {
533                                return true;
534                            }
535                            sb.Append(insertComma ? ", ..." : "...");
536                            return false;
537                        });
538                    }
539                    sb.Append('}');
540                    return sb.ToString();
541            }
542
543            [MethodImpl(MethodImplOptions.AggressiveInlining)]
544            public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
                ↪  knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
                ↪  knownElements);
545
546            [MethodImpl(MethodImplOptions.AggressiveInlining)]
547            public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
                ↪  LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
                ↪  Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
                ↪  sequenceLink, elementToString, insertComma, knownElements));
548
549            [MethodImpl(MethodImplOptions.AggressiveInlining)]
550            private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
                ↪  Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
                ↪  LinkIndex[] knownElements)
551            {
552                    var linksInSequence = new HashSet<ulong>(knownElements);
553                    var entered = new HashSet<ulong>();
554                    var sb = new StringBuilder();
555                    sb.Append('{');
556                    if (links.Exists(sequenceLink))
557                    {
558                        StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
559                            x => linksInSequence.Contains(x) || links.IsFullPoint(x),
                            ↪  entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
560                        {
561                            if (insertComma && sb.Length > 1)
562                            {
563                                sb.Append(',');
564                            }
565                            if (entered.Contains(element))
566                            {
567                                sb.Append('{');
568                                elementToString(sb, element);
569                                sb.Append('}');
570                            }
571                            else
572                            {
573                                elementToString(sb, element);
574                            }
575                            if (sb.Length < MaxSequenceFormatSize)
576                            {
577                                return true;
578                            }
579                            sb.Append(insertComma ? ", ..." : "...");
580                            return false;
```

```
581                        });
582                }
583            sb.Append('}');
584            return sb.ToString();
585        }
586
587        [MethodImpl(MethodImplOptions.AggressiveInlining)]
588        public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
589        {
590            return _sync.ExecuteReadOperation(() =>
591            {
592                if (sequence.Length > 0)
593                {
594                    Links.EnsureLinkExists(sequence);
595                    var results = new HashSet<ulong>();
596                    for (var i = 0; i < sequence.Length; i++)
597                    {
598                        AllUsagesCore(sequence[i], results);
599                    }
600                    var filteredResults = new List<ulong>();
601                    var linksInSequence = new HashSet<ulong>(sequence);
602                    foreach (var result in results)
603                    {
604                        var filterPosition = -1;
605                        StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
                                ↪  Links.Unsync.GetTarget,
606                            x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                                ↪  x =>
607                            {
608                                if (filterPosition == (sequence.Length - 1))
609                                {
610                                    return false;
611                                }
612                                if (filterPosition >= 0)
613                                {
614                                    if (x == sequence[filterPosition + 1])
615                                    {
616                                        filterPosition++;
617                                    }
618                                    else
619                                    {
620                                        return false;
621                                    }
622                                }
623                                if (filterPosition < 0)
624                                {
625                                    if (x == sequence[0])
626                                    {
627                                        filterPosition = 0;
628                                    }
629                                }
630                                return true;
631                            });
632                        if (filterPosition == (sequence.Length - 1))
633                        {
634                            filteredResults.Add(result);
635                        }
636                    }
637                    return filteredResults;
638                }
639                return new List<ulong>();
640            });
641        }
642
643        [MethodImpl(MethodImplOptions.AggressiveInlining)]
644        public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
645        {
646            return _sync.ExecuteReadOperation(() =>
647            {
648                if (sequence.Length > 0)
649                {
650                    Links.EnsureLinkExists(sequence);
651                    var results = new HashSet<ulong>();
652                    for (var i = 0; i < sequence.Length; i++)
653                    {
654                        AllUsagesCore(sequence[i], results);
655                    }
656                    var filteredResults = new HashSet<ulong>();
```

```csharp
                        var matcher = new Matcher(this, sequence, filteredResults, null);
                        matcher.AddAllPartialMatchedToResults(results);
                        return filteredResults;
                    }
                    return new HashSet<ulong>();
                });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
        ↪  params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);

                    var results = new HashSet<ulong>();
                    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(() =>
            {
                if (sequence.Length > 0)
                {
                    ILinksExtensions.EnsureLinkIsAnyOrExists(Links, 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);
```

```
735                        AllUsagesCore(last, lastResults);
736                        firstResults.IntersectWith(lastResults);
737                        //for (var i = 0; i < sequence.Length; i++)
738                        //    AllUsagesCore(sequence[i], results);
739                        var filteredResults = new HashSet<ulong>();
740                        var matcher = new Matcher(this, sequence, filteredResults, null);
741                        matcher.AddAllPartialMatchedToResults(firstResults);
742                        return filteredResults;
743                    }
744                    return new HashSet<ulong>();
745                });
746        }
747
748        [MethodImpl(MethodImplOptions.AggressiveInlining)]
749        public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
         ↪  IList<ulong> sequence)
750        {
751            return _sync.ExecuteReadOperation(() =>
752            {
753                if (sequence.Count > 0)
754                {
755                    Links.EnsureLinkExists(sequence);
756                    var results = new HashSet<LinkIndex>();
757                    //var nextResults = new HashSet<ulong>();
758                    //for (var i = 0; i < sequence.Length; i++)
759                    //{
760                    //    AllUsagesCore(sequence[i], nextResults);
761                    //    if (results.IsNullOrEmpty())
762                    //    {
763                    //        results = nextResults;
764                    //        nextResults = new HashSet<ulong>();
765                    //    }
766                    //    else
767                    //    {
768                    //        results.IntersectWith(nextResults);
769                    //        nextResults.Clear();
770                    //    }
771                    //}
772                    var collector1 = new AllUsagesCollector1(Links.Unsync, results);
773                    collector1.Collect(Links.Unsync.GetLink(sequence[0]));
774                    var next = new HashSet<ulong>();
775                    for (var i = 1; i < sequence.Count; i++)
776                    {
777                        var collector = new AllUsagesCollector1(Links.Unsync, next);
778                        collector.Collect(Links.Unsync.GetLink(sequence[i]));
779
780                        results.IntersectWith(next);
781                        next.Clear();
782                    }
783                    var filteredResults = new HashSet<ulong>();
784                    var matcher = new Matcher(this, sequence, filteredResults, null,
                     ↪  readAsElements);
785                    matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
                     ↪  x)); // OrderBy is a Hack
786                    return filteredResults;
787                }
788                return new HashSet<ulong>();
789            });
790        }
791
792        // Does not work
793        //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
         ↪  params ulong[] sequence)
794        //{
795        //    var visited = new HashSet<ulong>();
796        //    var results = new HashSet<ulong>();
797        //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
         ↪  true; }, readAsElements);
798        //    var last = sequence.Length - 1;
799        //    for (var i = 0; i < last; i++)
800        //    {
801        //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
802        //    }
803        //    return results;
804        //}
805
806        [MethodImpl(MethodImplOptions.AggressiveInlining)]
807        public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
```

```csharp
808         {
809             return _sync.ExecuteReadOperation(() =>
810             {
811                 if (sequence.Length > 0)
812                 {
813                     Links.EnsureLinkExists(sequence);
814                     //var firstElement = sequence[0];
815                     //if (sequence.Length == 1)
816                     //{
817                     //    //results.Add(firstElement);
818                     //    return results;
819                     //}
820                     //if (sequence.Length == 2)
821                     //{
822                     //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
823                     //    //if (doublet != Doublets.Links.Null)
824                     //    //    results.Add(doublet);
825                     //    return results;
826                     //}
827                     //var lastElement = sequence[sequence.Length - 1];
828                     //Func<ulong, bool> handler = x =>
829                     //{
830                     //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
        ↪  results.Add(x);
831                     //    return true;
832                     //};
833                     //if (sequence.Length >= 2)
834                     //    StepRight(handler, sequence[0], sequence[1]);
835                     //var last = sequence.Length - 2;
836                     //for (var i = 1; i < last; i++)
837                     //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
838                     //if (sequence.Length >= 3)
839                     //    StepLeft(handler, sequence[sequence.Length - 2],
        ↪  sequence[sequence.Length - 1]);
840                     //////if (sequence.Length == 1)
841                     //////{
842                     //////    throw new NotImplementedException(); // all sequences, containing
        ↪  this element?
843                     //////}
844                     //////if (sequence.Length == 2)
845                     //////{
846                     //////    var results = new List<ulong>();
847                     //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
848                     //////    return results;
849                     //////}
850                     //////var matches = new List<List<ulong>>();
851                     //////var last = sequence.Length - 1;
852                     //////for (var i = 0; i < last; i++)
853                     //////{
854                     //////    var results = new List<ulong>();
855                     //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
856                     //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
857                     //////    if (results.Count > 0)
858                     //////        matches.Add(results);
859                     //////    else
860                     //////        return results;
861                     //////    if (matches.Count == 2)
862                     //////    {
863                     //////        var merged = new List<ulong>();
864                     //////        for (var j = 0; j < matches[0].Count; j++)
865                     //////            for (var k = 0; k < matches[1].Count; k++)
866                     //////                CloseInnerConnections(merged.Add, matches[0][j],
        ↪  matches[1][k]);
867                     //////        if (merged.Count > 0)
868                     //////            matches = new List<List<ulong>> { merged };
869                     //////        else
870                     //////            return new List<ulong>();
871                     //////    }
872                     //////}
873                     //////if (matches.Count > 0)
874                     //////{
875                     //////    var usages = new HashSet<ulong>();
876                     //////    for (int i = 0; i < sequence.Length; i++)
877                     //////    {
878                     //////        AllUsagesCore(sequence[i], usages);
879                     //////    }
880                     //////    //for (int i = 0; i < matches[0].Count; i++)
```

```csharp
                //////      //      AllUsagesCore(matches[0][i], usages);
                //////      //usages.UnionWith(matches[0]);
                //////      return usages.ToList();
                //////}
                var firstLinkUsages = new HashSet<ulong>();
                AllUsagesCore(sequence[0], firstLinkUsages);
                firstLinkUsages.Add(sequence[0]);
                //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
                ↪  sequence[0] }; // or all sequences, containing this element?
                //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
                ↪  1).ToList();
                var results = new HashSet<ulong>();
                foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
                ↪  firstLinkUsages, 1))
                {
                    AllUsagesCore(match, results);
                }
                return results.ToList();
            }
            return new List<ulong>();
        });
    }

    /// <remarks>
    /// TODO: Может потробоваться ограничение на уровень глубины рекурсии
    /// </remarks>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public HashSet<ulong> AllUsages(ulong link)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            var usages = new HashSet<ulong>();
            AllUsagesCore(link, usages);
            return usages;
        });
    }

    // При сборе всех использований (последовательностей) можно сохранять обратный путь к
    ↪  той связи с которой начинался поиск (STTTSSSTT),
    // причём достаточно одного бита для хранения перехода влево или вправо
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private void AllUsagesCore(ulong link, HashSet<ulong> usages)
    {
        bool handler(ulong doublet)
        {
            if (usages.Add(doublet))
            {
                AllUsagesCore(doublet, usages);
            }
            return true;
        }
        Links.Unsync.Each(link, Constants.Any, handler);
        Links.Unsync.Each(Constants.Any, link, handler);
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public HashSet<ulong> AllBottomUsages(ulong link)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            var visits = new HashSet<ulong>();
            var usages = new HashSet<ulong>();
            AllBottomUsagesCore(link, visits, usages);
            return usages;
        });
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
    ↪  usages)
    {
        bool handler(ulong doublet)
        {
            if (visits.Add(doublet))
            {
                AllBottomUsagesCore(doublet, visits, usages);
            }
            return true;
        }
    }
```

```csharp
                }
                if (Links.Unsync.Count(Constants.Any, link) == 0)
                {
                    usages.Add(link);
                }
                else
                {
                    Links.Unsync.Each(link, Constants.Any, handler);
                    Links.Unsync.Each(Constants.Any, link, handler);
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
            {
                if (Options.UseSequenceMarker)
                {
                    var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                        ↪  Options.MarkedSequenceMatcher, symbol);
                    return counter.Count();
                }
                else
                {
                    var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                        ↪  symbol);
                    return counter.Count();
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
                ↪  LinkIndex> outerHandler)
            {
                bool handler(ulong doublet)
                {
                    if (usages.Add(doublet))
                    {
                        if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
                        {
                            return false;
                        }
                        if (!AllUsagesCore1(doublet, usages, outerHandler))
                        {
                            return false;
                        }
                    }
                    return true;
                }
                return Links.Unsync.Each(link, Constants.Any, handler)
                    && Links.Unsync.Each(Constants.Any, link, handler);
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void CalculateAllUsages(ulong[] totals)
            {
                var calculator = new AllUsagesCalculator(Links, totals);
                calculator.Calculate();
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void CalculateAllUsages2(ulong[] totals)
            {
                var calculator = new AllUsagesCalculator2(Links, totals);
                calculator.Calculate();
            }

            private class AllUsagesCalculator
            {
                private readonly SynchronizedLinks<ulong> _links;
                private readonly ulong[] _totals;

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
                {
                    _links = links;
                    _totals = totals;
                }

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
1030            public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
               ↪  CalculateCore);
1031
1032            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1033            private bool CalculateCore(ulong link)
1034            {
1035                if (_totals[link] == 0)
1036                {
1037                    var total = 1UL;
1038                    _totals[link] = total;
1039                    var visitedChildren = new HashSet<ulong>();
1040                    bool linkCalculator(ulong child)
1041                    {
1042                        if (link != child && visitedChildren.Add(child))
1043                        {
1044                            total += _totals[child] == 0 ? 1 : _totals[child];
1045                        }
1046                        return true;
1047                    }
1048                    _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1049                    _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1050                    _totals[link] = total;
1051                }
1052                return true;
1053            }
1054        }
1055
1056        private class AllUsagesCalculator2
1057        {
1058            private readonly SynchronizedLinks<ulong> _links;
1059            private readonly ulong[] _totals;
1060
1061            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1062            public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1063            {
1064                _links = links;
1065                _totals = totals;
1066            }
1067
1068            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1069            public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
               ↪  CalculateCore);
1070
1071            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1072            private bool IsElement(ulong link)
1073            {
1074                //_linksInSequence.Contains(link) ||
1075                return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
               ↪  link;
1076            }
1077
1078            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1079            private bool CalculateCore(ulong link)
1080            {
1081                // TODO: Проработать защиту от зацикливания
1082                // Основано на SequenceWalker.WalkLeft
1083                Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1084                Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1085                Func<ulong, bool> isElement = IsElement;
1086                void visitLeaf(ulong parent)
1087                {
1088                    if (link != parent)
1089                    {
1090                        _totals[parent]++;
1091                    }
1092                }
1093                void visitNode(ulong parent)
1094                {
1095                    if (link != parent)
1096                    {
1097                        _totals[parent]++;
1098                    }
1099                }
1100                var stack = new Stack();
1101                var element = link;
1102                if (isElement(element))
1103                {
1104                    visitLeaf(element);
1105                }
```

```csharp
                        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))
```

```csharp
                {
                    _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)
            {
```

```csharp
                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)
        {
```

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

```csharp
        [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;
```

```csharp
                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);
```

```csharp
                            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)]
```

```csharp
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;
                }
            }
```

```csharp
                    }
                }
                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

            enum PatternBlockType
            {
                Undefined,
                Gap,
                Elements
            }

            struct PatternBlock
            {
                public PatternBlockType Type;
                public long Start;
                public long Stop;
            }

            private readonly List<PatternBlock> _pattern;
            private int _patternPosition;
            private long _sequencePosition;

            #endregion

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
            ↪  HashSet<LinkIndex> results)
                : base(sequences.Links.Unsync, new DefaultStack<ulong>())
            {
                _sequences = sequences;
                _patternSequence = patternSequence;
                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                ↪  _sequences.Constants.Any && x != ZeroOrMany));
                _results = results;
                _pattern = CreateDetailedPattern();
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
            ↪  base.IsElement(link);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool PatternMatch(LinkIndex sequenceToMatch)
            {
                _patternPosition = 0;
                _sequencePosition = 0;
                foreach (var part in Walk(sequenceToMatch))
                {
                    if (!PatternMatchCore(part))
                    {
                        break;
                    }
                }
                return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
                ↪  - 1 && _pattern[_patternPosition].Start == 0);
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private List<PatternBlock> CreateDetailedPattern()
            {
                var pattern = new List<PatternBlock>();
                var patternBlock = new PatternBlock();
                for (var i = 0; i < _patternSequence.Length; i++)
                {
                    if (patternBlock.Type == PatternBlockType.Undefined)
```

```csharp
                    {
                        if (_patternSequence[i] == _sequences.Constants.Any)
                        {
                            patternBlock.Type = PatternBlockType.Gap;
                            patternBlock.Start = 1;
                            patternBlock.Stop = 1;
                        }
                        else if (_patternSequence[i] == ZeroOrMany)
                        {
                            patternBlock.Type = PatternBlockType.Gap;
                            patternBlock.Start = 0;
                            patternBlock.Stop = long.MaxValue;
                        }
                        else
                        {
                            patternBlock.Type = PatternBlockType.Elements;
                            patternBlock.Start = i;
                            patternBlock.Stop = i;
                        }
                    }
                    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;
            }
```

```csharp
            // 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
```

```csharp
                    }
                    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)
```

```
2022                {
2023                    foreach (var sequenceToMatch in sequencesToMatch)
2024                    {
2025                        if (PatternMatch(sequenceToMatch))
2026                        {
2027                            _results.Add(sequenceToMatch);
2028                        }
2029                    }
2030                }
2031            }

2033            #endregion
2034        }
2035    }
```

1.151 ./csharp/Platform.Data.Doublets/Sequences/Sequences.cs

```csharp
1    using System;
2    using System.Collections.Generic;
3    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;

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

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;

55           public SequencesOptions<LinkIndex> Options { get; }
56           public SynchronizedLinks<LinkIndex> Links { get; }
```

```csharp
        private readonly ISynchronization _sync;

        public LinksConstants<LinkIndex> Constants { get; }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
        {
            Links = links;
            _sync = links.SyncRoot;
            Options = options;
            Options.ValidateOptions();
            Options.InitOptions(Links);
            Constants = links.Constants;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
        ↪  SequencesOptions<LinkIndex>()) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsSequence(LinkIndex sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (Options.UseSequenceMarker)
                {
                    return Options.MarkedSequenceMatcher.IsMatched(sequence);
                }
                return !Links.Unsync.IsPartialPoint(sequence);
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex GetSequenceByElements(LinkIndex sequence)
        {
            if (Options.UseSequenceMarker)
            {
                return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
            }
            return sequence;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex GetSequenceElements(LinkIndex sequence)
        {
            if (Options.UseSequenceMarker)
            {
                var linkContents = new Link<ulong>(Links.GetLink(sequence));
                if (linkContents.Source == Options.SequenceMarkerLink)
                {
                    return linkContents.Target;
                }
                if (linkContents.Target == Options.SequenceMarkerLink)
                {
                    return linkContents.Source;
                }
            }
            return sequence;
        }

        #region Count

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Count(IList<LinkIndex> restrictions)
        {
            if (restrictions.IsNullOrEmpty())
            {
                return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
            }
            if (restrictions.Count == 1) // Первая связь это адрес
            {
                var sequenceIndex = restrictions[0];
                if (sequenceIndex == Constants.Null)
                {
                    return 0;
                }
                if (sequenceIndex == Constants.Any)
                {
                    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);
```

```csharp
                }
            if (sequenceRoot == default)
            {
                sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
            }
            if (Options.UseSequenceMarker)
            {
                return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
            }
            return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
        }

        #endregion

        #region Each

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<LinkIndex> Each(IList<LinkIndex> sequence)
        {
            var results = new List<LinkIndex>();
            var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
            Each(filler.AddFirstAndReturnConstant, sequence);
            return results;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
        ↪ restrictions)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (restrictions.IsNullOrEmpty())
                {
                    return Constants.Continue;
                }
                Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
                if (restrictions.Count == 1)
                {
                    var link = restrictions[0];
                    var any = Constants.Any;
                    if (link == any)
                    {
                        if (Options.UseSequenceMarker)
                        {
                            return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
                            ↪ Options.SequenceMarkerLink, any));
                        }
                        else
                        {
                            return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
                            ↪ any));
                        }
                    }
                    if (Options.UseSequenceMarker)
                    {
                        var sequenceLinkValues = Links.Unsync.GetLink(link);
                        if (sequenceLinkValues[Constants.SourcePart] ==
                        ↪ Options.SequenceMarkerLink)
                        {
                            link = sequenceLinkValues[Constants.TargetPart];
                        }
                    }
                    var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
                    sequence[0] = link;
                    return handler(sequence);
                }
                else if (restrictions.Count == 2)
                {
                    throw new NotImplementedException();
                }
                else if (restrictions.Count == 3)
                {
                    return Links.Unsync.Each(handler, restrictions);
                }
                else
                {
                    var sequence = restrictions.SkipFirst();
                    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);
369              while (firstTarget != left && firstTarget != upStep)
370              {
371                  upStep = firstTarget;
372                  firstTarget = Links.Unsync.GetTarget(upStep);
373              }
374              if (firstTarget == left)
375              {
376                  return handler(new LinkAddress<LinkIndex>(stepFrom));
377              }
378              return Constants.Continue;
379          }
380
381          #endregion
382
383          #region Update
384
385          [MethodImpl(MethodImplOptions.AggressiveInlining)]
386          public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
387          {
388              var sequence = restrictions.SkipFirst();
389              var newSequence = substitution.SkipFirst();
390              if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
391              {
392                  return Constants.Null;
393              }
394              if (sequence.IsNullOrEmpty())
395              {
396                  return Create(substitution);
397              }
398              if (newSequence.IsNullOrEmpty())
399              {
400                  Delete(restrictions);
401                  return Constants.Null;
402              }
403              return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
404              {
405                  ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
406                  Links.EnsureLinkExists(newSequence);
407                  return UpdateCore(sequence, newSequence);
408              }));
409          }
410
411          [MethodImpl(MethodImplOptions.AggressiveInlining)]
412          private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
413          {
414              LinkIndex bestVariant;
415              if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
                ↪   !sequence.EqualTo(newSequence))
416              {
417                  bestVariant = CompactCore(newSequence);
418              }
419              else
420              {
421                  bestVariant = CreateCore(newSequence);
422              }
423              // TODO: Check all options only ones before loop execution
424              // Возможно нужно две версии Each, возвращающий фактические последовательности и с
                ↪   маркером,
```

```csharp
425                // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
         ↪        можно получить имея только фактические последовательности.
426                foreach (var variant in Each(sequence))
427                {
428                    if (variant != bestVariant)
429                    {
430                        UpdateOneCore(variant, bestVariant);
431                    }
432                }
433                return bestVariant;
434            }
435
436            [MethodImpl(MethodImplOptions.AggressiveInlining)]
437            private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
438            {
439                if (Options.UseGarbageCollection)
440                {
441                    var sequenceElements = GetSequenceElements(sequence);
442                    var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
443                    var sequenceLink = GetSequenceByElements(sequenceElements);
444                    var newSequenceElements = GetSequenceElements(newSequence);
445                    var newSequenceLink = GetSequenceByElements(newSequenceElements);
446                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
447                    {
448                        if (sequenceLink != Constants.Null)
449                        {
450                            Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
451                        }
452                        Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
453                    }
454                    ClearGarbage(sequenceElementsContents.Source);
455                    ClearGarbage(sequenceElementsContents.Target);
456                }
457                else
458                {
459                    if (Options.UseSequenceMarker)
460                    {
461                        var sequenceElements = GetSequenceElements(sequence);
462                        var sequenceLink = GetSequenceByElements(sequenceElements);
463                        var newSequenceElements = GetSequenceElements(newSequence);
464                        var newSequenceLink = GetSequenceByElements(newSequenceElements);
465                        if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
466                        {
467                            if (sequenceLink != Constants.Null)
468                            {
469                                Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
470                            }
471                            Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
472                        }
473                    }
474                    else
475                    {
476                        if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
477                        {
478                            Links.Unsync.MergeAndDelete(sequence, newSequence);
479                        }
480                    }
481                }
482            }
483
484            #endregion
485
486            #region Delete
487
488            [MethodImpl(MethodImplOptions.AggressiveInlining)]
489            public void Delete(IList<LinkIndex> restrictions)
490            {
491                _sync.ExecuteWriteOperation(() =>
492                {
493                    var sequence = restrictions.SkipFirst();
494                    // TODO: Check all options only ones before loop execution
495                    foreach (var linkToDelete in Each(sequence))
496                    {
497                        DeleteOneCore(linkToDelete);
498                    }
499                });
500            }
501
```

```csharp
        [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
            {
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(link);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    if (Options.UseCascadeDelete || CountUsages(link) == 0)
                    {
                        if (sequenceLink != Constants.Null)
                        {
                            Links.Unsync.Delete(sequenceLink);
                        }
                        Links.Unsync.Delete(link);
                    }
                }
                else
                {
                    if (Options.UseCascadeDelete || CountUsages(link) == 0)
                    {
                        Links.Unsync.Delete(link);
                    }
                }
            }
        }

        #endregion

        #region Compactification

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CompactAll()
        {
            _sync.ExecuteWriteOperation(() =>
            {
                var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
                for (int i = 0; i < sequences.Count; i++)
                {
                    var sequence = this.ToList(sequences[i]);
                    Compact(sequence.ShiftRight());
                }
            });
        }

        /// <remarks>
        /// bestVariant можно выбирать по максимальному числу использований,
        /// но балансированный позволяет гарантировать уникальность (если есть возможность,
        /// гарантировать его использование в других местах).
        /// 
        /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Compact(IList<LinkIndex> sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return Constants.Null;
                }
```

```csharp
                    Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
                    return CompactCore(sequence);
                });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex CompactCore(IList<LinkIndex> sequence) => UpdateCore(sequence,
          sequence);

        #endregion

        #region Garbage Collection

        /// <remarks>
        /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
          определить извне или в унаследованном классе
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
          !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ClearGarbage(LinkIndex link)
        {
            if (IsGarbage(link))
            {
                var contents = new Link<ulong>(Links.GetLink(link));
                Links.Unsync.Delete(link);
                ClearGarbage(contents.Source);
                ClearGarbage(contents.Target);
            }
        }

        #endregion

        #region Walkers

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var links = Links.Unsync;
                foreach (var part in Options.Walker.Walk(sequence))
                {
                    if (!handler(part))
                    {
                        return false;
                    }
                }
                return true;
            });
        }

        public class Matcher : RightSequenceWalker<LinkIndex>
        {
            private readonly Sequences _sequences;
            private readonly IList<LinkIndex> _patternSequence;
            private readonly HashSet<LinkIndex> _linksInSequence;
            private readonly HashSet<LinkIndex> _results;
            private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
            private readonly HashSet<LinkIndex> _readAsElements;
            private int _filterPosition;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
              HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
              HashSet<LinkIndex> readAsElements = null)
                : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
            {
                _sequences = sequences;
                _patternSequence = patternSequence;
                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                  _links.Constants.Any && x != ZeroOrMany));
                _results = results;
                _stopableHandler = stopableHandler;
                _readAsElements = readAsElements;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
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                 {
```

```csharp
                if (!PartialMatchCore(part))
                {
                    break;
                }
            }
            return _filterPosition == _patternSequence.Count - 1;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool PartialMatchCore(LinkIndex element)
        {
            if (_filterPosition == (_patternSequence.Count - 1))
            {
                return false; // Нашлось
            }
            if (_filterPosition >= 0)
            {
                if (element == _patternSequence[_filterPosition + 1])
                {
                    _filterPosition++;
                }
                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);
```

```
807                    }
808                }
809            }
810        }
811
812        #endregion
813    }
814 }
```

1.152 ./csharp/Platform.Data.Doublets/Sequences/SequencesExtensions.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences
8  {
9      public static class SequencesExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static TLink Create<TLink>(this ILinks<TLink> sequences, IList<TLink[]>
           ↪   groupedSequence)
13         {
14             var finalSequence = new TLink[groupedSequence.Count];
15             for (var i = 0; i < finalSequence.Length; i++)
16             {
17                 var part = groupedSequence[i];
18                 finalSequence[i] = part.Length == 1 ? part[0] :
                   ↪   sequences.Create(part.ShiftRight());
19             }
20             return sequences.Create(finalSequence.ShiftRight());
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
25         {
26             var list = new List<TLink>();
27             var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
28             sequences.Each(filler.AddSkipFirstAndReturnConstant, new
               ↪   LinkAddress<TLink>(sequence));
29             return list;
30         }
31     }
32 }
```

1.153 ./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs

```
1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Collections.Stacks;
5  using Platform.Converters;
6  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8  using Platform.Data.Doublets.Sequences.Converters;
9  using Platform.Data.Doublets.Sequences.Walkers;
10 using Platform.Data.Doublets.Sequences.Indexes;
11 using Platform.Data.Doublets.Sequences.CriterionMatchers;
12 using System.Runtime.CompilerServices;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
       ↪   ILinks<TLink> must contain GetConstants function.
19     {
20         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪   EqualityComparer<TLink>.Default;
21
22         public TLink SequenceMarkerLink
23         {
24             [MethodImpl(MethodImplOptions.AggressiveInlining)]
25             get;
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             set;
28         }
29
30         public bool UseCascadeUpdate
31         {
```

```csharp
            [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)]
```

```csharp
113                 get;
114                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115                 set;
116             }
117
118         public ISequenceWalker<TLink> Walker
119         {
120                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
121                 get;
122                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
123                 set;
124             }
125
126         public bool ReadFullSequence
127         {
128                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129                 get;
130                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
131                 set;
132             }
133
134         // TODO: Реализовать компактификацию при чтении
135         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
136         //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             }
207         }
208     }
209 }
```

1.154   ./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.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.Sequences.Walkers
7  {
8      public interface ISequenceWalker<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         IEnumerable<TLink> Walk(TLink sequence);
12     }
13 }
```

1.155   ./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Stacks;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Walkers
9  {
10     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
    ↪  isElement) : base(links, stack, isElement) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
    ↪  links.IsPartialPoint) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override TLink GetNextElementAfterPop(TLink element) =>
    ↪  _links.GetSource(element);
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override TLink GetNextElementAfterPush(TLink element) =>
    ↪  _links.GetTarget(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override IEnumerable<TLink> WalkContents(TLink element)
26         {
27             var links = _links;
28             var parts = links.GetLink(element);
29             var start = links.Constants.SourcePart;
30             for (var i = parts.Count - 1; i >= start; i--)
31             {
```

```csharp
                    var part = parts[i];
                    if (IsElement(part))
                    {
                        yield return part;
                    }
                }
            }
        }
    }
```

1.156 ./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs

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

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

//#define USEARRAYPOOL
#if USEARRAYPOOL
using Platform.Collections;
#endif

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;

        private readonly Func<TLink, bool> _isElement;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
            base(links) => _isElement = isElement;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
            _links.IsPartialPoint;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink[] ToArray(TLink sequence)
        {
            var length = 1;
            var array = new TLink[length];
            array[0] = sequence;
            if (_isElement(sequence))
            {
                return array;
            }
            bool hasElements;
            do
            {
                length *= 2;
#if USEARRAYPOOL
                var nextArray = ArrayPool.Allocate<ulong>(length);
#else
                var nextArray = new TLink[length];
#endif
                hasElements = false;
                for (var i = 0; i < array.Length; i++)
                {
                    var candidate = array[i];
                    if (_equalityComparer.Equals(array[i], default))
                    {
                        continue;
                    }
                    var doubletOffset = i * 2;
                    if (_isElement(candidate))
                    {
                        nextArray[doubletOffset] = candidate;
                    }
                    else
                    {
                        var links = _links;
                        var link = links.GetLink(candidate);
                        var linkSource = links.GetSource(link);
                        var linkTarget = links.GetTarget(link);
```

```csharp
                            nextArray[doubletOffset] = linkSource;
                            nextArray[doubletOffset + 1] = linkTarget;
                            if (!hasElements)
                            {
                                hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
                            }
                        }
                    }
#if USEARRAYPOOL
                    if (array.Length > 1)
                    {
                        ArrayPool.Free(array);
                    }
#endif
                    array = nextArray;
                }
                while (hasElements);
                var filledElementsCount = CountFilledElements(array);
                if (filledElementsCount == array.Length)
                {
                    return array;
                }
                else
                {
                    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;
        }
    }
}
```

1.157 ./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs

```csharp
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)]
```

```csharp
16      public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
    ↪  stack, links.IsPartialPoint) { }
17
18      [MethodImpl(MethodImplOptions.AggressiveInlining)]
19      protected override TLink GetNextElementAfterPop(TLink element) =>
    ↪  _links.GetTarget(element);
20
21      [MethodImpl(MethodImplOptions.AggressiveInlining)]
22      protected override TLink GetNextElementAfterPush(TLink element) =>
    ↪  _links.GetSource(element);
23
24      [MethodImpl(MethodImplOptions.AggressiveInlining)]
25      protected override IEnumerable<TLink> WalkContents(TLink element)
26      {
27          var parts = _links.GetLink(element);
28          for (var i = _links.Constants.SourcePart; i < parts.Count; i++)
29          {
30              var part = parts[i];
31              if (IsElement(part))
32              {
33                  yield return part;
34              }
35          }
36      }
37  }
38 }
```

1.158 ./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs

```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Stacks;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Walkers
9  {
10     public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceWalker<TLink>
11     {
12         private readonly IStack<TLink> _stack;
13         private readonly Func<TLink, bool> _isElement;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
    ↪  isElement) : base(links)
17         {
18             _stack = stack;
19             _isElement = isElement;
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
    ↪  stack, links.IsPartialPoint) { }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public IEnumerable<TLink> Walk(TLink sequence)
27         {
28             _stack.Clear();
29             var element = sequence;
30             if (IsElement(element))
31             {
32                 yield return element;
33             }
34             else
35             {
36                 while (true)
37                 {
38                     if (IsElement(element))
39                     {
40                         if (_stack.IsEmpty)
41                         {
42                             break;
43                         }
44                         element = _stack.Pop();
45                         foreach (var output in WalkContents(element))
46                         {
47                             yield return output;
48                         }
```

```
49                        element = GetNextElementAfterPop(element);
50                    }
51                    else
52                    {
53                        _stack.Push(element);
54                        element = GetNextElementAfterPush(element);
55                    }
56                }
57            }
58        }
59
60        [MethodImpl(MethodImplOptions.AggressiveInlining)]
61        protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
62
63        [MethodImpl(MethodImplOptions.AggressiveInlining)]
64        protected abstract TLink GetNextElementAfterPop(TLink element);
65
66        [MethodImpl(MethodImplOptions.AggressiveInlining)]
67        protected abstract TLink GetNextElementAfterPush(TLink element);
68
69        [MethodImpl(MethodImplOptions.AggressiveInlining)]
70        protected abstract IEnumerable<TLink> WalkContents(TLink element);
71    }
72 }
```

1.159 ./csharp/Platform.Data.Doublets/Stacks/Stack.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Stacks;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Stacks
8  {
9      public class Stack<TLink> : LinksOperatorBase<TLink>, IStack<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
12
13         private readonly TLink _stack;
14
15         public bool IsEmpty
16         {
17             [MethodImpl(MethodImplOptions.AggressiveInlining)]
18             get => _equalityComparer.Equals(Peek(), _stack);
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public Stack(ILinks<TLink> links, TLink stack) : base(links) => _stack = stack;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         private TLink GetStackMarker() => _links.GetSource(_stack);
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         private TLink GetTop() => _links.GetTarget(_stack);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public TLink Peek() => _links.GetTarget(GetTop());
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public TLink Pop()
35         {
36             var element = Peek();
37             if (!_equalityComparer.Equals(element, _stack))
38             {
39                 var top = GetTop();
40                 var previousTop = _links.GetSource(top);
41                 _links.Update(_stack, GetStackMarker(), previousTop);
42                 _links.Delete(top);
43             }
44             return element;
45         }
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
            ↪  _links.GetOrCreate(GetTop(), element));
49     }
50 }
```

1.160 ./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs

```csharp
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;
        }
    }
}
```

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

```csharp
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)]
```

```csharp
        public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
        ↪  IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
        ↪  restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
        ↪  SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
        ↪  substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
        ↪  Unsync.Update);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Delete(IList<TLinkAddress> restrictions) =>
        ↪  SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);

        //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
        ↪  IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
        //{
        //    if (restriction != null && substitution != null &&
        ↪  !substitution.EqualTo(restriction))
        //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
        ↪  substitution, substitutedHandler, Unsync.Trigger);

        //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
        ↪  substitutedHandler, Unsync.Trigger);
        //}
    }
}
```

1.162 ./csharp/Platform.Data.Doublets/Time/DateTimeToLongRawNumberSequenceConverter.cs

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

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

namespace Platform.Data.Doublets.Time
{
    public class DateTimeToLongRawNumberSequenceConverter<TLink> : IConverter<DateTime, TLink>
    {
        private readonly IConverter<long, TLink> _int64ToLongRawNumberConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DateTimeToLongRawNumberSequenceConverter(IConverter<long, TLink>
        ↪  int64ToLongRawNumberConverter) => _int64ToLongRawNumberConverter =
        ↪  int64ToLongRawNumberConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(DateTime source) =>
        ↪  _int64ToLongRawNumberConverter.Convert(source.ToFileTimeUtc());
    }
}
```

1.163 ./csharp/Platform.Data.Doublets/Time/LongRawNumberSequenceToDateTimeConverter.cs

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

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

namespace Platform.Data.Doublets.Time
{
    public class LongRawNumberSequenceToDateTimeConverter<TLink> : IConverter<TLink, DateTime>
    {
        private readonly IConverter<TLink, long> _longRawNumberConverterToInt64;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LongRawNumberSequenceToDateTimeConverter(IConverter<TLink, long>
        ↪  longRawNumberConverterToInt64) => _longRawNumberConverterToInt64 =
        ↪  longRawNumberConverterToInt64;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DateTime Convert(TLink source) =>
        ↪  DateTime.FromFileTimeUtc(_longRawNumberConverterToInt64.Convert(source));
    }
}
```

```csharp
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;
25              }
26              var constants = links.Constants;
27              for (var i = 0; i < sequence.Length; i++)
28              {
29                  if (sequence[i] == constants.Any)
30                  {
31                      return true;
32                  }
33              }
34              return false;
35          }
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
                   Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
                   false)
39          {
40              var sb = new StringBuilder();
41              var visited = new HashSet<ulong>();
42              links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
                       innerSb.Append(link.Index), renderIndex, renderDebug);
43              return sb.ToString();
44          }
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          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)
48          {
49              var sb = new StringBuilder();
50              var visited = new HashSet<ulong>();
51              links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
                       renderDebug);
52              return sb.ToString();
53          }
54
55          [MethodImpl(MethodImplOptions.AggressiveInlining)]
56          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)
57          {
58              if (sb == null)
59              {
60                  throw new ArgumentNullException(nameof(sb));
61              }
62              if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
                       Constants.Itself)
63              {
64                  return;
65              }
66              if (links.Exists(linkIndex))
67              {
```

```
68              if (visited.Add(linkIndex))
69              {
70                  sb.Append('(');
71                  var link = new Link<ulong>(links.GetLink(linkIndex));
72                  if (renderIndex)
73                  {
74                      sb.Append(link.Index);
75                      sb.Append(':');
76                  }
77                  if (link.Source == link.Index)
78                  {
79                      sb.Append(link.Index);
80                  }
81                  else
82                  {
83                      var source = new Link<ulong>(links.GetLink(link.Source));
84                      if (isElement(source))
85                      {
86                          appendElement(sb, source);
87                      }
88                      else
89                      {
90                          links.AppendStructure(sb, visited, source.Index, isElement,
   ↪  appendElement, renderIndex);
91                      }
92                  }
93                  sb.Append(' ');
94                  if (link.Target == link.Index)
95                  {
96                      sb.Append(link.Index);
97                  }
98                  else
99                  {
100                     var target = new Link<ulong>(links.GetLink(link.Target));
101                     if (isElement(target))
102                     {
103                         appendElement(sb, target);
104                     }
105                     else
106                     {
107                         links.AppendStructure(sb, visited, target.Index, isElement,
   ↪  appendElement, renderIndex);
108                     }
109                 }
110                 sb.Append(')');
111             }
112             else
113             {
114                 if (renderDebug)
115                 {
116                     sb.Append('*');
117                 }
118                 sb.Append(linkIndex);
119             }
120         }
121         else
122         {
123             if (renderDebug)
124             {
125                 sb.Append('~');
126             }
127             sb.Append(linkIndex);
128         }
129     }
130   }
131 }
```

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

```
1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Runtime.CompilerServices;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Platform.Disposables;
9  using Platform.Timestamps;
10 using Platform.Unsafe;
11 using Platform.IO;
```

```csharp
using Platform.Data.Doublets.Decorators;
using Platform.Exceptions;

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

namespace Platform.Data.Doublets
{
    public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
    {
        /// <remarks>
        /// Альтернативные варианты хранения трансформации (элемента транзакции):
        ///
        /// private enum TransitionType
        /// {
        ///     Creation,
        ///     UpdateOf,
        ///     UpdateTo,
        ///     Deletion
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public ulong TransactionId;
        ///     public UniqueTimestamp Timestamp;
        ///     public TransactionItemType Type;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
        /// }
        ///
        /// Или
        ///
        /// public struct TransitionHeader
        /// {
        ///     public ulong TransactionIdCombined;
        ///     public ulong TimestampCombined;
        ///
        ///     public ulong TransactionId
        ///     {
        ///         get
        ///         {
        ///             return (ulong) mask &amp; TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public UniqueTimestamp Timestamp
        ///     {
        ///         get
        ///         {
        ///             return (UniqueTimestamp)mask &amp; TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public TransactionItemType Type
        ///     {
        ///         get
        ///         {
        ///             // Использовать по одному биту из TransactionId и Timestamp,
        ///             // для значения в 2 бита, которое представляет тип операции
        ///             throw new NotImplementedException();
        ///         }
        ///     }
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public TransitionHeader Header;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
        /// }
        ///
        /// </remarks>
        public struct Transition : IEquatable<Transition>
        {
            public static readonly long Size = Structure<Transition>.Size;

            public readonly ulong TransactionId;
```

```csharp
        public readonly Link<ulong> Before;
        public readonly Link<ulong> After;
        public readonly Timestamp Timestamp;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
        ↪   transactionId, Link<ulong> before, Link<ulong> after)
        {
            TransactionId = transactionId;
            Before = before;
            After = after;
            Timestamp = uniqueTimestampFactory.Create();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
        ↪   transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
        ↪   before, default) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
        ↪   transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
        ↪   }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
        ↪   {After}";

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is Transition transition ?
        ↪   Equals(transition) : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (TransactionId, Before, After,
        ↪   Timestamp).GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(Transition other) => TransactionId == other.TransactionId &&
        ↪   Before == other.Before && After == other.After && Timestamp == other.Timestamp;

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

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

    /// <remarks>
    /// Другие варианты реализации транзакций (атомарности):
    ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
    ↪   Target)) и индексов.
    ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
    ↪   потребуется решить вопрос
    ///         со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
    ↪   пересечениями идентификаторов.
    ///
    /// Где хранить промежуточный список транзакций?
    ///
    /// В оперативной памяти:
    ///   Минусы:
    ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
    ///     так как нужно отдельно выделять память под список трансформаций.
    ///     2. Выделенной оперативной памяти может не хватить, в том случае,
    ///     если транзакция использует слишком много трансформаций.
    ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
    ///         -> Максимальный размер списка трансформаций можно ограничить / задать
    ↪   константой.
    ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
    ↪   создавая задержку.
    ///
    /// На жёстком диске:
    ///   Минусы:
    ///     1. Длительный отклик, на запись каждой трансформации.
    ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
    ///         -> Это может решаться упаковкой/исключением дублирующих операций.
    ///         -> Также это может решаться тем, что короткие транзакции вообще
```

```
///                  не будут записываться в случае отката.
///       3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
↪ операции (трансформации)
///          будут записаны в лог.
///
/// </remarks>
public class Transaction : DisposableBase
{
    private readonly Queue<Transition> _transitions;
    private readonly UInt64LinksTransactionsLayer _layer;
    public bool IsCommitted { get; private set; }
    public bool IsReverted { get; private set; }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public Transaction(UInt64LinksTransactionsLayer layer)
    {
        _layer = layer;
        if (_layer._currentTransactionId != 0)
        {
            throw new NotSupportedException("Nested transactions not supported.");
        }
        IsCommitted = false;
        IsReverted = false;
        _transitions = new Queue<Transition>();
        SetCurrentTransaction(layer, this);
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public void Commit()
    {
        EnsureTransactionAllowsWriteOperations(this);
        while (_transitions.Count > 0)
        {
            var transition = _transitions.Dequeue();
            _layer._transitions.Enqueue(transition);
        }
        _layer._lastCommitedTransactionId = _layer._currentTransactionId;
        IsCommitted = true;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private void Revert()
    {
        EnsureTransactionAllowsWriteOperations(this);
        var transitionsToRevert = new Transition[_transitions.Count];
        _transitions.CopyTo(transitionsToRevert, 0);
        for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
        {
            _layer.RevertTransition(transitionsToRevert[i]);
        }
        IsReverted = true;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
    ↪ Transaction transaction)
    {
        layer._currentTransactionId = layer._lastCommitedTransactionId + 1;
        layer._currentTransactionTransitions = transaction._transitions;
        layer._currentTransaction = transaction;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
    {
        if (transaction.IsReverted)
        {
            throw new InvalidOperationException("Transation is reverted.");
        }
        if (transaction.IsCommitted)
        {
            throw new InvalidOperationException("Transation is commited.");
        }
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    protected override void Dispose(bool manual, bool wasDisposed)
    {
        if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
```

```csharp
                    {
                        if (!IsCommitted && !IsReverted)
                        {
                            Revert();
                        }
                        _layer.ResetCurrentTransation();
                    }
                }
            }

        public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);

        private readonly string _logAddress;
        private readonly FileStream _log;
        private readonly Queue<Transition> _transitions;
        private readonly UniqueTimestampFactory _uniqueTimestampFactory;
        private Task _transitionsPusher;
        private Transition _lastCommitedTransition;
        private ulong _currentTransactionId;
        private Queue<Transition> _currentTransactionTransitions;
        private Transaction _currentTransaction;
        private ulong _lastCommitedTransactionId;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
            : base(links)
        {
            if (string.IsNullOrWhiteSpace(logAddress))
            {
                throw new ArgumentNullException(nameof(logAddress));
            }
            // В первой строке файла хранится последняя закоммиченную транзакцию.
            // При запуске это используется для проверки удачного закрытия файла лога.
            // In the first line of the file the last committed transaction is stored.
            // On startup, this is used to check that the log file is successfully closed.
            var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
            var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
            if (!lastCommitedTransition.Equals(lastWrittenTransition))
            {
                Dispose();
                throw new NotSupportedException("Database is damaged, autorecovery is not
                    supported yet.");
            }
            if (lastCommitedTransition == default)
            {
                FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
            }
            _lastCommitedTransition = lastCommitedTransition;
            // TODO: Think about a better way to calculate or store this value
            var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
            _lastCommitedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
                x.TransactionId) : 0;
            _uniqueTimestampFactory = new UniqueTimestampFactory();
            _logAddress = logAddress;
            _log = FileHelpers.Append(logAddress);
            _transitions = new Queue<Transition>();
            _transitionsPusher = new Task(TransitionsPusher);
            _transitionsPusher.Start();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override ulong Create(IList<ulong> restrictions)
        {
            var createdLinkIndex = _links.Create();
            var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
            CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                default, createdLink));
            return createdLinkIndex;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
        {
            var linkIndex = restrictions[_constants.IndexPart];
            var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
            linkIndex = _links.Update(restrictions, substitution);
```

```
305         var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
306         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
            ↪  beforeLink, afterLink));
307         return linkIndex;
308     }
309
310     [MethodImpl(MethodImplOptions.AggressiveInlining)]
311     public override void Delete(IList<ulong> restrictions)
312     {
313         var link = restrictions[_constants.IndexPart];
314         var deletedLink = new Link<ulong>(_links.GetLink(link));
315         _links.Delete(link);
316         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
            ↪  deletedLink, default));
317     }
318
319     [MethodImpl(MethodImplOptions.AggressiveInlining)]
320     private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
        ↪  _transitions;
321
322     [MethodImpl(MethodImplOptions.AggressiveInlining)]
323     private void CommitTransition(Transition transition)
324     {
325         if (_currentTransaction != null)
326         {
327             Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
328         }
329         var transitions = GetCurrentTransitions();
330         transitions.Enqueue(transition);
331     }
332
333     [MethodImpl(MethodImplOptions.AggressiveInlining)]
334     private void RevertTransition(Transition transition)
335     {
336         if (transition.After.IsNull()) // Revert Deletion with Creation
337         {
338             _links.Create();
339         }
340         else if (transition.Before.IsNull()) // Revert Creation with Deletion
341         {
342             _links.Delete(transition.After.Index);
343         }
344         else // Revert Update
345         {
346             _links.Update(new[] { transition.After.Index, transition.Before.Source,
                ↪  transition.Before.Target });
347         }
348     }
349
350     [MethodImpl(MethodImplOptions.AggressiveInlining)]
351     private void ResetCurrentTransation()
352     {
353         _currentTransactionId = 0;
354         _currentTransactionTransitions = null;
355         _currentTransaction = null;
356     }
357
358     [MethodImpl(MethodImplOptions.AggressiveInlining)]
359     private void PushTransitions()
360     {
361         if (_log == null || _transitions == null)
362         {
363             return;
364         }
365         for (var i = 0; i < _transitions.Count; i++)
366         {
367             var transition = _transitions.Dequeue();
368
369             _log.Write(transition);
370             _lastCommitedTransition = transition;
371         }
372     }
373
374     [MethodImpl(MethodImplOptions.AggressiveInlining)]
375     private void TransitionsPusher()
376     {
377         while (!Disposable.IsDisposed && _transitionsPusher != null)
378         {
```

```
379                    Thread.Sleep(DefaultPushDelay);
380                    PushTransitions();
381                }
382            }
383
384        [MethodImpl(MethodImplOptions.AggressiveInlining)]
385        public Transaction BeginTransaction() => new Transaction(this);
386
387        [MethodImpl(MethodImplOptions.AggressiveInlining)]
388        private void DisposeTransitions()
389        {
390            try
391            {
392                var pusher = _transitionsPusher;
393                if (pusher != null)
394                {
395                    _transitionsPusher = null;
396                    pusher.Wait();
397                }
398                if (_transitions != null)
399                {
400                    PushTransitions();
401                }
402                _log.DisposeIfPossible();
403                FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
404            }
405            catch (Exception ex)
406            {
407                ex.Ignore();
408            }
409        }
410
411        #region DisposalBase
412
413        [MethodImpl(MethodImplOptions.AggressiveInlining)]
414        protected override void Dispose(bool manual, bool wasDisposed)
415        {
416            if (!wasDisposed)
417            {
418                DisposeTransitions();
419            }
420            base.Dispose(manual, wasDisposed);
421        }
422
423        #endregion
424    }
425 }
```

1.166   ./csharp/Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs

```
1  using System.Runtime.CompilerServices;
2  using Platform.Converters;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Unicode
7  {
8      public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
         ↪  IConverter<char, TLink>
9      {
10         private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
           ↪  UncheckedConverter<char, TLink>.Default;
11
12         private readonly IConverter<TLink> _addressToNumberConverter;
13         private readonly TLink _unicodeSymbolMarker;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
           ↪  addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
17         {
18             _addressToNumberConverter = addressToNumberConverter;
19             _unicodeSymbolMarker = unicodeSymbolMarker;
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public TLink Convert(char source)
24         {
25             var unaryNumber =
               ↪  _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
26             return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
27         }
```

```
 28            }
 29    }
```

1.167 ./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs

```csharp
 1   using System.Collections.Generic;
 2   using System.Runtime.CompilerServices;
 3   using Platform.Converters;
 4   using Platform.Data.Doublets.Sequences.Indexes;
 5
 6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 7
 8   namespace Platform.Data.Doublets.Unicode
 9   {
10       public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
         ↪  IConverter<string, TLink>
11       {
12           private readonly IConverter<string, IList<TLink>> _stringToUnicodeSymbolListConverter;
13           private readonly IConverter<IList<TLink>, TLink> _unicodeSymbolListToSequenceConverter;
14
15           [MethodImpl(MethodImplOptions.AggressiveInlining)]
16           public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
         ↪  IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
         ↪  unicodeSymbolListToSequenceConverter) : base(links)
17           {
18               _stringToUnicodeSymbolListConverter = stringToUnicodeSymbolListConverter;
19               _unicodeSymbolListToSequenceConverter = unicodeSymbolListToSequenceConverter;
20           }
21
22           [MethodImpl(MethodImplOptions.AggressiveInlining)]
23           public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
         ↪  IList<TLink>> stringToUnicodeSymbolListConverter, ISequenceIndex<TLink> index,
         ↪  IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
         ↪  unicodeSequenceMarker)
24               : this(links, stringToUnicodeSymbolListConverter, new
                 ↪  UnicodeSymbolsListToUnicodeSequenceConverter<TLink>(links, index,
                 ↪  listToSequenceLinkConverter, unicodeSequenceMarker)) { }
25
26           [MethodImpl(MethodImplOptions.AggressiveInlining)]
27           public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
         ↪  charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
         ↪  TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker)
28               : this(links, new
                 ↪  StringToUnicodeSymbolsListConverter<TLink>(charToUnicodeSymbolConverter), index,
                 ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }
29
30           [MethodImpl(MethodImplOptions.AggressiveInlining)]
31           public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
         ↪  charToUnicodeSymbolConverter, IConverter<IList<TLink>, TLink>
         ↪  listToSequenceLinkConverter, TLink unicodeSequenceMarker)
32               : this(links, charToUnicodeSymbolConverter, new Unindex<TLink>(),
                 ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }
33
34           [MethodImpl(MethodImplOptions.AggressiveInlining)]
35           public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
         ↪  IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
         ↪  listToSequenceLinkConverter, TLink unicodeSequenceMarker)
36               : this(links, stringToUnicodeSymbolListConverter, new Unindex<TLink>(),
                 ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }
37
38           [MethodImpl(MethodImplOptions.AggressiveInlining)]
39           public TLink Convert(string source)
40           {
41               var elements = _stringToUnicodeSymbolListConverter.Convert(source);
42               return _unicodeSymbolListToSequenceConverter.Convert(elements);
43           }
44       }
45   }
```

1.168 ./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSymbolsListConverter.cs

```csharp
 1   using System.Collections.Generic;
 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.Unicode
 8   {
 9       public class StringToUnicodeSymbolsListConverter<TLink> : IConverter<string, IList<TLink>>
10       {
```

```csharp
        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;
        }
    }
}
```

1.169 ./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs

```csharp
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.");
                    }
                }
            }
```

```csharp
        }

        // 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)
            {
```

```csharp
                    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)]
        public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
        {
            var result = new List<ulong[]>();
            var offset = 0;
            while (offset < array.Length)
            {
                var relativeLength = 1;
                if (array[offset] <= LastCharLink)
                {
                    var currentCategory =
                        ↪ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
                    var absoluteLength = offset + relativeLength;
                    while (absoluteLength < array.Length &&
                            array[absoluteLength] <= LastCharLink &&
                            currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
                            ↪ array[absoluteLength])))
                    {
                        relativeLength++;
                        absoluteLength++;
                    }
                }
                else
                {
                    var absoluteLength = offset + relativeLength;
                    while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
                    {
                        relativeLength++;
                        absoluteLength++;
                    }
                }
                // copy array
                var innerSequence = new ulong[relativeLength];
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = array[i];
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }
    }
}
```

1.170 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs

```csharp
using System;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
using System.Text;

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

namespace Platform.Data.Doublets.Unicode
```

```csharp
{
    public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
      IConverter<TLink, string>
    {
        private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
        private readonly ISequenceWalker<TLink> _sequenceWalker;
        private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
          unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
          IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
        {
            _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
            _sequenceWalker = sequenceWalker;
            _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public string Convert(TLink source)
        {
            if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
            {
                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();
        }
    }
}
```

1.171 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs

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

1.172 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs

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

1.173 ./csharp/Platform.Data.Doublets.Tests/DefaultSequenceAppenderTests.cs

```csharp
using System.Collections.Generic;
using Platform.Collections.Stacks;
using Platform.Data.Doublets.Memory;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.HeightProviders;
using Platform.Data.Numbers.Raw;
using Platform.Interfaces;
using Platform.Memory;
using Platform.Numbers;
using Xunit;
using Xunit.Abstractions;
using TLink = System.UInt64;

namespace Platform.Data.Doublets.Tests
{
    public class DefaultSequenceAppenderTests
    {
        private readonly ITestOutputHelper _output;

        public DefaultSequenceAppenderTests(ITestOutputHelper output)
        {
            _output = output;
        }
        public static ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());

        public static ILinks<TLink> CreateLinks<TLink>(string dataDBFilename)
        {
            var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
                true);
            return new UnitedMemoryLinks<TLink>(new
                FileMappedResizableDirectMemory(dataDBFilename),
                UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
                IndexTreeType.Default);
        }
```

```
32
33         public class ValueCriterionMatcher<TLink> : ICriterionMatcher<TLink>
34         {
35             public readonly ILinks<TLink> Links;
36             public readonly TLink Marker;
37             public ValueCriterionMatcher(ILinks<TLink> links, TLink marker)
38             {
39                 Links = links;
40                 Marker = marker;
41             }
42
43             public bool IsMatched(TLink link) =>
       ↪         EqualityComparer<TLink>.Default.Equals(Links.GetSource(link), Marker);
44         }
45
46         [Fact]
47         public void AppendArrayBug()
48         {
49             ILinks<TLink> links = CreateLinks();
50             TLink zero = default;
51             var markerIndex = Arithmetic.Increment(zero);
52             var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
53             var sequence = links.Create();
54             sequence = links.Update(sequence, meaningRoot, sequence);
55             var appendant = links.Create();
56             appendant = links.Update(appendant, meaningRoot, appendant);
57             ValueCriterionMatcher<TLink> valueCriterionMatcher = new(links, meaningRoot);
58             DefaultSequenceRightHeightProvider<ulong> defaultSequenceRightHeightProvider =
       ↪         new(links, valueCriterionMatcher);
59             DefaultSequenceAppender<TLink> defaultSequenceAppender = new(links, new
       ↪         DefaultStack<ulong>(), defaultSequenceRightHeightProvider);
60             var newArray = defaultSequenceAppender.Append(sequence, appendant);
61             var output = links.FormatStructure(newArray, link => link.IsFullPoint(), true);
62             Assert.Equal("(4:(2:1 2) (3:1 3))", output);
63         }
64     }
65 }
```

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

```
1  using System;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Memory;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.Memory.United.Generic;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public unsafe static class GenericLinksTests
11     {
12         [Fact]
13         public static void CRUDTest()
14         {
15             Using<byte>(links => links.TestCRUDOperations());
16             Using<ushort>(links => links.TestCRUDOperations());
17             Using<uint>(links => links.TestCRUDOperations());
18             Using<ulong>(links => links.TestCRUDOperations());
19         }
20
21         [Fact]
22         public static void RawNumbersCRUDTest()
23         {
24             Using<byte>(links => links.TestRawNumbersCRUDOperations());
25             Using<ushort>(links => links.TestRawNumbersCRUDOperations());
26             Using<uint>(links => links.TestRawNumbersCRUDOperations());
27             Using<ulong>(links => links.TestRawNumbersCRUDOperations());
28         }
29
30         [Fact]
31         public static void MultipleRandomCreationsAndDeletionsTest()
32         {
33             Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
       ↪         MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
       ↪         implementation of tree cuts out 5 bits from the address space.
34             Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
       ↪         stMultipleRandomCreationsAndDeletions(100));
35             Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
       ↪         MultipleRandomCreationsAndDeletions(100));
```

```csharp
                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>>());
            }
        }
    }
}
```

1.175 ./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs

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

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

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

1.177 ./csharp/Platform.Data.Doublets.Tests/Numbers/Raw/BigIntegerConvertersTests.cs

```csharp
using System.Numerics;
using Platform.Data.Doublets.Memory;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Numbers.Raw;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Numbers.Raw;
using Platform.Memory;
using Xunit;
using TLink = System.UInt64;

namespace Platform.Data.Doublets.Tests.Numbers.Raw
{
    public class BigIntegerConvertersTests
    {
        public ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());

        public ILinks<TLink> CreateLinks<TLink>(string dataDbFilename)
        {
```

```csharp
                var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
                ↪    true);
                return new UnitedMemoryLinks<TLink>(new
                ↪    FileMappedResizableDirectMemory(dataDbFilename),
                ↪    UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
                ↪    IndexTreeType.Default);
            }
            [Fact]
            public void Test()
            {
                var links = CreateLinks();
                BigInteger bigInt = new(1);
                AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
                RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
                BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
                BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪    = new(links, addressToRawNumberConverter, listToSequenceConverter);
                RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪    = new(links, numberToAddressConverter);
                var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInt);
                var bigIntFromSequence =
                ↪    rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
                Assert.Equal(bigInt, bigIntFromSequence);
            }

            [Fact]
            public void Test64BitNumber()
            {
                var links = CreateLinks();
                BigInteger bigInt = new(ulong.MaxValue);
                AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
                RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
                BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
                BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪    = new(links, addressToRawNumberConverter, listToSequenceConverter);
                RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪    = new(links, numberToAddressConverter);
                var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInt);
                var bigIntFromSequence =
                ↪    rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
                Assert.Equal(bigInt, bigIntFromSequence);
            }

            [Fact]
            public void Test65BitNumber()
            {
                var links = CreateLinks();
                BigInteger bigInt = new(ulong.MaxValue);
                bigInt *= 2;
                AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
                RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
                BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
                BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪    = new(links, addressToRawNumberConverter, listToSequenceConverter);
                RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪    = new(links, numberToAddressConverter);
                var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInt);
                var bigIntFromSequence =
                ↪    rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
                Assert.Equal(bigInt, bigIntFromSequence);
            }
        }
    }
```

1.178 ./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs

```csharp
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;
```

```csharp
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);
```

```
 95              var unaryNumberToAddressConverter = new
     ↪   UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
 96              var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
 97              var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
     ↪   frequencyMarker, unaryOne, unaryNumberIncrementer);
 98              var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
     ↪   frequencyPropertyMarker, frequencyMarker);
 99              var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
     ↪   frequencyPropertyOperator, frequencyIncrementer);
100              var linkToItsFrequencyNumberConverter = new
     ↪   LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
     ↪   unaryNumberToAddressConverter);
101              var sequenceToItsLocalElementLevelsConverter = new
     ↪   SequenceToItsLocalElementLevelsConverter<ulong>(links,
     ↪   linkToItsFrequencyNumberConverter);
102              var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
     ↪   sequenceToItsLocalElementLevelsConverter);
103
104              var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
     ↪   Walker = new LeveledSequenceWalker<ulong>(links) });
105
106              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()
```

```csharp
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);
168                var index = new
                 ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
169                var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque⌋
                 ↪  ncyNumberConverter<ulong>(linkFrequenciesCache);
170                var sequenceToItsLocalElementLevelsConverter = new
                 ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                 ↪  linkToItsFrequencyNumberConverter);
171                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                 ↪  sequenceToItsLocalElementLevelsConverter);
172
173                var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
                 ↪  (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
174
175                var unicodeSequencesOptions = new SequencesOptions<ulong>()
176                {
177                    UseSequenceMarker = true,
178                    SequenceMarkerLink = unicodeSequenceMarker,
179                    UseIndex = true,
180                    Index = index,
181                    LinksToSequenceConverter = optimalVariantConverter,
182                    Walker = walker,
183                    UseGarbageCollection = true
184                };
185
186                var unicodeSequences = new Sequences.Sequences(new
                 ↪  SynchronizedLinks<ulong>(links), unicodeSequencesOptions);
187
188                // Create some sequences
189                var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
                 ↪  StringSplitOptions.RemoveEmptyEntries);
190                var arrays = strings.Select(x => x.Select(y =>
                 ↪  addressToNumberConverter.Convert(y)).ToArray()).ToArray();
191                for (int i = 0; i < arrays.Length; i++)
192                {
193                    unicodeSequences.Create(arrays[i].ShiftRight());
194                }
195
196                var linksCountAfterCreation = links.Count();
197
198                // get list of sequences links
199                // for each sequence link
200                //   create new sequence version
201                //   if new sequence is not the same as sequence link
202                //     delete sequence link
203                //     collect garbadge
204                unicodeSequences.CompactAll();
205
206                var linksCountAfterCompactification = links.Count();
207
208                Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
209            }
210        }
211    }
212 }
```

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

```csharp
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;
```

```
13          [Fact]
14          public static void BasicFileMappedMemoryTest()
15          {
16              var tempFilename = Path.GetTempFileName();
17              using (var memoryAdapter = new UInt64UnitedMemoryLinks(tempFilename))
18              {
19                  memoryAdapter.TestBasicMemoryOperations();
20              }
21              File.Delete(tempFilename);
22          }
23
24          [Fact]
25          public static void BasicHeapMemoryTest()
26          {
27              using (var memory = new
                ↪  HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
28              using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
                ↪  UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
29              {
30                  memoryAdapter.TestBasicMemoryOperations();
31              }
32          }
33
34          private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
35          {
36              var link = memoryAdapter.Create();
37              memoryAdapter.Delete(link);
38          }
39
40          [Fact]
41          public static void NonexistentReferencesHeapMemoryTest()
42          {
43              using (var memory = new
                ↪  HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
44              using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
                ↪  UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
45              {
46                  memoryAdapter.TestNonexistentReferences();
47              }
48          }
49
50          private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
51          {
52              var link = memoryAdapter.Create();
53              memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
54              var resultLink = _constants.Null;
55              memoryAdapter.Each(foundLink =>
56              {
57                  resultLink = foundLink[_constants.IndexPart];
58                  return _constants.Break;
59              }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
60              Assert.True(resultLink == link);
61              Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
62              memoryAdapter.Delete(link);
63          }
64      }
65  }
```

1.181   ./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs

```
1   using Xunit;
2   using Platform.Scopes;
3   using Platform.Memory;
4   using Platform.Data.Doublets.Decorators;
5   using Platform.Reflection;
6   using Platform.Data.Doublets.Memory.United.Generic;
7   using Platform.Data.Doublets.Memory.United.Specific;
8
9   namespace Platform.Data.Doublets.Tests
10  {
11      public static class ScopeTests
12      {
13          [Fact]
14          public static void SingleDependencyTest()
15          {
16              using (var scope = new Scope())
17              {
18                  scope.IncludeAssemblyOf<IMemory>();
19                  var instance = scope.Use<IDirectMemory>();
20                  Assert.IsType<HeapResizableDirectMemory>(instance);
```

```
21                }
22            }
23
24        [Fact]
25        public static void CascadeDependencyTest()
26        {
27            using (var scope = new Scope())
28            {
29                scope.Include<TemporaryFileMappedResizableDirectMemory>();
30                scope.Include<UInt64UnitedMemoryLinks>();
31                var instance = scope.Use<ILinks<ulong>>();
32                Assert.IsType<UInt64UnitedMemoryLinks>(instance);
33            }
34        }
35
36        [Fact(Skip = "Would be fixed later.")]
37        public static void FullAutoResolutionTest()
38        {
39            using (var scope = new Scope(autoInclude: true, autoExplore: true))
40            {
41                var instance = scope.Use<UInt64Links>();
42                Assert.IsType<UInt64Links>(instance);
43            }
44        }
45
46        [Fact]
47        public static void TypeParametersTest()
48        {
49            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  UnitedMemoryLinks<ulong>>>())
50            {
51                var links = scope.Use<ILinks<ulong>>();
52                Assert.IsType<UnitedMemoryLinks<ulong>>(links);
53            }
54        }
55    }
56 }
```

1.182 ./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Collections;
7  using Platform.Collections.Arrays;
8  using Platform.Random;
9  using Platform.IO;
10 using Platform.Singletons;
11 using Platform.Data.Doublets.Sequences;
12 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
13 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
14 using Platform.Data.Doublets.Sequences.Converters;
15 using Platform.Data.Doublets.Unicode;
16
17 namespace Platform.Data.Doublets.Tests
18 {
19    public static class SequencesTests
20    {
21        private static readonly LinksConstants<ulong> _constants =
            ↪  Default<LinksConstants<ulong>>.Instance;
22
23        static SequencesTests()
24        {
25            // Trigger static constructor to not mess with perfomance measurements
26            _ = BitString.GetBitMaskFromIndex(1);
27        }
28
29        [Fact]
30        public static void CreateAllVariantsTest()
31        {
32            const long sequenceLength = 8;
33
34            using (var scope = new TempLinksTestScope(useSequences: true))
35            {
36                var links = scope.Links;
37                var sequences = scope.Sequences;
38
39                var sequence = new ulong[sequenceLength];
40                for (var i = 0; i < sequenceLength; i++)
```

```csharp
                {
                    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();
```

```
120            //for (int i = 0; i < createResults.Length; i++)
121            //    sequences.Create(createResults[i]);
122
123            var sw0 = Stopwatch.StartNew();
124            var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125
126            var sw1 = Stopwatch.StartNew();
127            var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128
129            var sw2 = Stopwatch.StartNew();
130            var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131
132            var sw3 = Stopwatch.StartNew();
133            var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
134
135            var intersection0 = createResults.Intersect(searchResults0).ToList();
136            Assert.True(intersection0.Count == searchResults0.Count);
137            Assert.True(intersection0.Count == createResults.Length);
138
139            var intersection1 = createResults.Intersect(searchResults1).ToList();
140            Assert.True(intersection1.Count == searchResults1.Count);
141            Assert.True(intersection1.Count == createResults.Length);
142
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            }
```

```csharp
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();
221                    //Global.Trash = createResultsStrings;
222
223                    var partialSequence = new ulong[sequenceLength - 2];
224
225                    Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
226
227                    var sw1 = Stopwatch.StartNew();
228                    var searchResults1 =
                        ↪   sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
229
230                    var sw2 = Stopwatch.StartNew();
231                    var searchResults2 =
                        ↪   sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
232
233                    //var sw3 = Stopwatch.StartNew();
234                    //var searchResults3 =
                        ↪   sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
235
236                    var sw4 = Stopwatch.StartNew();
237                    var searchResults4 =
                        ↪   sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
238
239                    //Global.Trash = searchResults3;
240
241                    //var searchResults1Strings = searchResults1.Select(x => x + ": " +
                        ↪   sequences.FormatSequence(x)).ToList();
242                    //Global.Trash = searchResults1Strings;
243
244                    var intersection1 = createResults.Intersect(searchResults1).ToList();
245                    Assert.True(intersection1.Count == createResults.Length);
246
247                    var intersection2 = createResults.Intersect(searchResults2).ToList();
248                    Assert.True(intersection2.Count == createResults.Length);
249
250                    var intersection4 = createResults.Intersect(searchResults4).ToList();
251                    Assert.True(intersection4.Count == createResults.Length);
252
253                    for (var i = 0; i < sequenceLength; i++)
254                    {
255                        links.Delete(sequence[i]);
256                    }
257                }
258            }
259
260            [Fact]
261            public static void BalancedPartialVariantsSearchTest()
262            {
263                const long sequenceLength = 200;
264
265                using (var scope = new TempLinksTestScope(useSequences: true))
266                {
267                    var links = scope.Links;
268                    var sequences = scope.Sequences;
269
270                    var sequence = new ulong[sequenceLength];
271                    for (var i = 0; i < sequenceLength; i++)
272                    {
```

```
273                      sequence[i] = links.Create();
274                  }
275
276              var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
277
278              var balancedVariant = balancedVariantConverter.Convert(sequence);
279
280              var partialSequence = new ulong[sequenceLength - 2];
281
282              Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
283
284              var sw1 = Stopwatch.StartNew();
285              var searchResults1 =
     ↪    sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
286
287              var sw2 = Stopwatch.StartNew();
288              var searchResults2 =
     ↪    sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
289
290              Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
291
292              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
             ↪  version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
382
383  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?
384
385  [![чёрное пространство, белое
     ↪  пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
     ↪  ""чёрное пространство, белое пространство"")](https://raw.githubusercontent.com/Konard/Links
     ↪  Platform/master/doc/Intro/1.png)
386
387  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?
388
389  [![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)
390
391  А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
     ↪  так? Инверсия? Отражение? Сумма?
392
393  [![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)
394
395  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?
396
397  [![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)
398
399  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?
400
```

[![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)

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?

[![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)

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?

[![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)

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)

...

```
433    [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anima
    ↪    tion-500.gif
    ↪    ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↪    -animation-500.gif)";
434
435        private static readonly string _exampleLoremIpsumText =
436            @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
             ↪    incididunt ut labore et dolore magna aliqua.
437    Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
    ↪    consequat.";
438
439        [Fact]
440        public static void CompressionTest()
441        {
442            using (var scope = new TempLinksTestScope(useSequences: true))
443            {
444                var links = scope.Links;
445                var sequences = scope.Sequences;
446
447                var e1 = links.Create();
448                var e2 = links.Create();
449
450                var sequence = new[]
451                {
452                    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))
```

```csharp
            {
                scope1.Links.Unsync.UseUnicode();
                scope2.Links.Unsync.UseUnicode();
                scope3.Links.Unsync.UseUnicode();

                var balancedVariantConverter1 = new
                    BalancedVariantConverter<ulong>(scope1.Links.Unsync);
                var totalSequenceSymbolFrequencyCounter = new
                    TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
                var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
                    totalSequenceSymbolFrequencyCounter);
                var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
                    balancedVariantConverter1, linkFrequenciesCache1,
                    doInitialFrequenciesIncrement: false);

                //var compressor2 = scope2.Sequences;
                var compressor3 = scope3.Sequences;

                var constants = Default<LinksConstants<ulong>>.Instance;

                var sequences = compressor3;
                //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 FrequencyPropertyOperator<ulong>(links,
                    frequencyPropertyMarker, frequencyMarker);
                //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                    frequencyPropertyOperator, frequencyIncrementer);
                //var linkToItsFrequencyNumberConverter = new
                    LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    unaryNumberToAddressConverter);

                var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
                    totalSequenceSymbolFrequencyCounter);

                var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ncyNumberConverter<ulong>(linkFrequenciesCache3);

                var sequenceToItsLocalElementLevelsConverter = new
                    SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
                    linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new
                    OptimalVariantConverter<ulong>(scope3.Links.Unsync,
                    sequenceToItsLocalElementLevelsConverter);

                var compressed1 = new ulong[arrays.Length];
                var compressed2 = new ulong[arrays.Length];
                var compressed3 = new ulong[arrays.Length];

                var START = 0;
                var END = arrays.Length;

                //for (int i = START; i < END; i++)
                //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);

                var initialCount1 = scope2.Links.Unsync.Count();

                var sw1 = Stopwatch.StartNew();

                for (int i = START; i < END; i++)
                {
                    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
                    compressed1[i] = compressor1.Convert(arrays[i]);
                }

                var elapsed1 = sw1.Elapsed;

                var balancedVariantConverter2 = new
                    BalancedVariantConverter<ulong>(scope2.Links.Unsync);
```

```csharp
                var initialCount2 = scope2.Links.Unsync.Count();

                var sw2 = Stopwatch.StartNew();

                for (int i = START; i < END; i++)
                {
                    compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
                }

                var elapsed2 = sw2.Elapsed;

                for (int i = START; i < END; i++)
                {
                    linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
                }

                var initialCount3 = scope3.Links.Unsync.Count();

                var sw3 = Stopwatch.StartNew();

                for (int i = START; i < END; i++)
                {
                    //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
                    compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
                }

                var elapsed3 = sw3.Elapsed;

                Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
                ↪ Optimal variant: {elapsed3}");

                // Assert.True(elapsed1 > elapsed2);

                // Checks
                for (int i = START; i < END; i++)
                {
                    var sequence1 = compressed1[i];
                    var sequence2 = compressed2[i];
                    var sequence3 = compressed3[i];

                    var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                    ↪ scope1.Links.Unsync);

                    var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                    ↪ scope2.Links.Unsync);

                    var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
                    ↪ scope3.Links.Unsync);

                    var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
                    ↪ link.IsPartialPoint());
                    var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
                    ↪ link.IsPartialPoint());
                    var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
                    ↪ link.IsPartialPoint());

                    //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                    ↪ arrays[i].Length > 3)
                    //    Assert.False(structure1 == structure2);
                    //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
                    ↪ arrays[i].Length > 3)
                    //    Assert.False(structure3 == structure2);

                    Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
                    Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
                }

                Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
                ↪ totalCharacters);
                Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
                ↪ totalCharacters);
                Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
                ↪ totalCharacters);
```

```csharp
            Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
                totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
                totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
                totalCharacters}");

            Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
                scope2.Links.Unsync.Count() - initialCount2);
            Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
                scope2.Links.Unsync.Count() - initialCount2);

            var duplicateProvider1 = new
                DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
            var duplicateProvider2 = new
                DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
            var duplicateProvider3 = new
                DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);

            var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
            var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
            var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);

            var duplicates1 = duplicateCounter1.Count();

            ConsoleHelpers.Debug("------");

            var duplicates2 = duplicateCounter2.Count();

            ConsoleHelpers.Debug("------");

            var duplicates3 = duplicateCounter3.Count();

            Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");

            linkFrequenciesCache1.ValidateFrequencies();
            linkFrequenciesCache3.ValidateFrequencies();
        }
    }

    [Fact]
    public static void CompressionStabilityTest()
    {
        // TODO: Fix bug (do a separate test)
        //const ulong minNumbers = 0;
        //const ulong maxNumbers = 1000;

        const ulong minNumbers = 10000;
        const ulong maxNumbers = 12500;

        var strings = new List<string>();

        for (ulong i = minNumbers; i < maxNumbers; i++)
        {
            strings.Add(i.ToString());
        }

        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 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
            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;

            // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
```

```csharp
            // Stability issue starts at 10001 or 11000
            //for (int i = START; i < END; i++)
            //{
            //    var first = compressor1.Compress(arrays[i]);
            //    var second = compressor1.Compress(arrays[i]);

            //    if (first == second)
            //        compressed1[i] = first;
            //    else
            //    {
            //        // TODO: Find a solution for this case
            //    }
            //}

            for (int i = START; i < END; i++)
            {
                var first = compressor1.Create(arrays[i].ShiftRight());
                var second = compressor1.Create(arrays[i].ShiftRight());

                if (first == second)
                {
                    compressed1[i] = first;
                }
                else
                {
                    // TODO: Find a solution for this case
                }
            }

            var elapsed1 = sw1.Elapsed;

            var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);

            var sw2 = Stopwatch.StartNew();

            for (int i = START; i < END; i++)
            {
                var first = balancedVariantConverter.Convert(arrays[i]);
                var second = balancedVariantConverter.Convert(arrays[i]);

                if (first == second)
                {
                    compressed2[i] = first;
                }
            }

            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);

                    //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                    ↪  link.IsPartialPoint());
                    //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                    ↪  link.IsPartialPoint());

                    //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                    ↪  arrays[i].Length > 3)
                    //    Assert.False(structure1 == structure2);

                    Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
                }
            }
```

```csharp
                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}");

                Assert.True(scope1.Links.Count() <= scope2.Links.Count());

                //compressor1.ValidateFrequencies();
            }
        }

        [Fact]
        public static void RandomNumbersCompressionQualityTest()
        {
            const ulong N = 500;

            //const ulong minNumbers = 10000;
            //const ulong maxNumbers = 20000;

            //var strings = new List<string>();

            //for (ulong i = 0; i < N; i++)
            //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
            //    maxNumbers).ToString());

            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}");
```

```csharp
                    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();
```

```csharp
                    }

                    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);
                    }
```

```
989             for (var i = 0; i < sequenceLength; i++)
990             {
991                 links.Delete(sequence[i]);
992             }
993         }
994     }
995   }
996 }
```

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

```csharp
1  using System;
2  using Xunit;
3  using Platform.Memory;
4  using Platform.Data.Doublets.Memory.Split.Generic;
5  using Platform.Data.Doublets.Memory;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public unsafe static class SplitMemoryGenericLinksTests
10     {
11         [Fact]
12         public static void CRUDTest()
13         {
14             Using<byte>(links => links.TestCRUDOperations());
15             Using<ushort>(links => links.TestCRUDOperations());
16             Using<uint>(links => links.TestCRUDOperations());
17             Using<ulong>(links => links.TestCRUDOperations());
18         }
19
20         [Fact]
21         public static void RawNumbersCRUDTest()
22         {
23             UsingWithExternalReferences<byte>(links => links.TestRawNumbersCRUDOperations());
24             UsingWithExternalReferences<ushort>(links => links.TestRawNumbersCRUDOperations());
25             UsingWithExternalReferences<uint>(links => links.TestRawNumbersCRUDOperations());
26             UsingWithExternalReferences<ulong>(links => links.TestRawNumbersCRUDOperations());
27         }
28
29         [Fact]
30         public static void MultipleRandomCreationsAndDeletionsTest()
31         {
32             Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
                ↪ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
                ↪ implementation of tree cuts out 5 bits from the address space.
33             Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
                ↪ stMultipleRandomCreationsAndDeletions(100));
34             Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
                ↪ MultipleRandomCreationsAndDeletions(100));
35             Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
                ↪ tMultipleRandomCreationsAndDeletions(100));
36         }
37
38         private static void Using<TLink>(Action<ILinks<TLink>> action)
39         {
40             using (var dataMemory = new HeapResizableDirectMemory())
41             using (var indexMemory = new HeapResizableDirectMemory())
42             using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory))
43             {
44                 action(memory);
45             }
46         }
47
48         private static void UsingWithExternalReferences<TLink>(Action<ILinks<TLink>> action)
49         {
50             var contants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
51             using (var dataMemory = new HeapResizableDirectMemory())
52             using (var indexMemory = new HeapResizableDirectMemory())
53             using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory,
                ↪ SplitMemoryLinks<TLink>.DefaultLinksSizeStep, contants))
54             {
55                 action(memory);
56             }
57         }
58     }
59 }
```

1.184 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs

```csharp
1  using System;
2  using Xunit;
```

```csharp
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Specific;
using TLink = System.UInt32;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class SplitMemoryUInt32LinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            UsingWithExternalReferences(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip
            ↪  leRandomCreationsAndDeletions(500));
        }

        private static void Using(Action<ILinks<TLink>> action)
        {
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory))
            {
                action(memory);
            }
        }

        private static void UsingWithExternalReferences(Action<ILinks<TLink>> action)
        {
            var contants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory,
            ↪  UInt32SplitMemoryLinks.DefaultLinksSizeStep, contants))
            {
                action(memory);
            }
        }
    }
}
```

1.185   ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt64LinksTests.cs

```csharp
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));
```

```
27              }
28
29          private static void Using(Action<ILinks<TLink>> action)
30          {
31              using (var dataMemory = new HeapResizableDirectMemory())
32              using (var indexMemory = new HeapResizableDirectMemory())
33              using (var memory = new UInt64SplitMemoryLinks(dataMemory, indexMemory))
34              {
35                  action(memory);
36              }
37          }
38
39          private static void UsingWithExternalReferences(Action<ILinks<TLink>> action)
40          {
41              var contants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
42              using (var dataMemory = new HeapResizableDirectMemory())
43              using (var indexMemory = new HeapResizableDirectMemory())
44              using (var memory = new UInt64SplitMemoryLinks(dataMemory, indexMemory,
                  ↪ UInt64SplitMemoryLinks.DefaultLinksSizeStep, contants))
45              {
46                  action(memory);
47              }
48          }
49      }
50  }
```

1.186 ./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs

```
1   using System.IO;
2   using Platform.Disposables;
3   using Platform.Data.Doublets.Sequences;
4   using Platform.Data.Doublets.Decorators;
5   using Platform.Data.Doublets.Memory.United.Specific;
6   using Platform.Data.Doublets.Memory.Split.Specific;
7   using Platform.Memory;
8
9   namespace Platform.Data.Doublets.Tests
10  {
11      public class TempLinksTestScope : DisposableBase
12      {
13          public ILinks<ulong> MemoryAdapter { get; }
14          public SynchronizedLinks<ulong> Links { get; }
15          public Sequences.Sequences Sequences { get; }
16          public string TempFilename { get; }
17          public string TempTransactionLogFilename { get; }
18          private readonly bool _deleteFiles;
19
20          public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
                  ↪ useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
                  ↪ useLog) { }
21
22          public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
                  ↪ true, bool useSequences = false, bool useLog = false)
23          {
24              _deleteFiles = deleteFiles;
25              TempFilename = Path.GetTempFileName();
26              TempTransactionLogFilename = Path.GetTempFileName();
27              //var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
28              var coreMemoryAdapter = new UInt64SplitMemoryLinks(new
                  ↪ FileMappedResizableDirectMemory(TempFilename), new
                  ↪ FileMappedResizableDirectMemory(Path.ChangeExtension(TempFilename, "indexes")),
                  ↪ UInt64SplitMemoryLinks.DefaultLinksSizeStep, new LinksConstants<ulong>(),
                  ↪ Memory.IndexTreeType.Default, useLinkedList: true);
29              MemoryAdapter = useLog ? (ILinks<ulong>)new
                  ↪ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
                  ↪ coreMemoryAdapter;
30              Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
31              if (useSequences)
32              {
33                  Sequences = new Sequences.Sequences(Links, sequencesOptions);
34              }
35          }
36
37          protected override void Dispose(bool manual, bool wasDisposed)
38          {
39              if (!wasDisposed)
40              {
41                  Links.Unsync.DisposeIfPossible();
42                  if (_deleteFiles)
43                  {
```

```csharp
                        DeleteFiles();
                    }
                }
            }

            public void DeleteFiles()
            {
                File.Delete(TempFilename);
                File.Delete(TempTransactionLogFilename);
            }
        }
    }
```

1.187 ./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs

```csharp
using System.Collections.Generic;
using Xunit;
using Platform.Ranges;
using Platform.Numbers;
using Platform.Random;
using Platform.Setters;
using Platform.Converters;

namespace Platform.Data.Doublets.Tests
{
    public static class TestExtensions
    {
        public static void TestCRUDOperations<T>(this ILinks<T> links)
        {
            var constants = links.Constants;

            var equalityComparer = EqualityComparer<T>.Default;

            var zero = default(T);
            var one = Arithmetic.Increment(zero);

            // Create Link
            Assert.True(equalityComparer.Equals(links.Count(), zero));

            var setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter.Result, constants.Null));

            var linkAddress = links.Create();

            var link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(link.Count == 3);
            Assert.True(equalityComparer.Equals(link.Index, linkAddress));
            Assert.True(equalityComparer.Equals(link.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link.Target, constants.Null));

            Assert.True(equalityComparer.Equals(links.Count(), one));

            // Get first link
            setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);

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

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

            link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(equalityComparer.Equals(link.Source, linkAddress));
            Assert.True(equalityComparer.Equals(link.Target, linkAddress));

            // Update link to reference null (prepare for delete)
            var updated = links.Update(linkAddress, constants.Null, constants.Null);

            Assert.True(equalityComparer.Equals(updated, linkAddress));

            link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(equalityComparer.Equals(link.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link.Target, constants.Null));

            // Delete link
            links.Delete(linkAddress);
```

```
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)
172                    {
173                        var linksAddressRange = new Range<ulong>(1, linksCount);
174                        TLink source = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA⌋
                       ↪  ddressRange));
175                        TLink target = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA⌋
                       ↪  ddressRange));
                       ↪  //-V3086
176                        var resultLink = links.GetOrCreate(source, target);
177                        if (comparer.Compare(resultLink,
                       ↪  uInt64ToAddressConverter.Convert(linksCount)) > 0)
178                        {
179                            created++;
180                        }
181                    }
182                    else
183                    {
184                        links.Create();
185                        created++;
186                    }
187                }
188                Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
189                for (var i = 0; i < N; i++)
190                {
191                    TLink link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
192                    if (links.Exists(link))
193                    {
194                        links.Delete(link);
195                        deleted++;
196                    }
197                }
198                Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
199            }
200        }
201    }
202 }
```

1.188 ./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.IO;
5  using System.Text;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Xunit;
9  using Platform.Disposables;
10 using Platform.Ranges;
11 using Platform.Random;
12 using Platform.Timestamps;
13 using Platform.Reflection;
14 using Platform.Singletons;
15 using Platform.Scopes;
16 using Platform.Counters;
17 using Platform.Diagnostics;
18 using Platform.IO;
```

```csharp
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);
            }
        }

        [Fact]
        public static void CascadeUpdateTest()
        {
            var itself = _constants.Itself;
            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;

                var l1 = links.Create();
                var l2 = links.Create();

                l2 = links.Update(l2, l2, l1, l2);

                links.CreateAndUpdate(l2, itself);
                links.CreateAndUpdate(l2, itself);

                l2 = links.Update(l2, l1);

                links.Delete(l2);

                Global.Trash = links.Count();

                links.Unsync.DisposeIfPossible(); // Close links to access log

                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    e.TempTransactionLogFilename);
            }
        }

        [Fact]
        public static void BasicTransactionLogTest()
        {
            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;
                var l1 = links.Create();
                var l2 = links.Create();

                Global.Trash = links.Update(l2, l2, l1, l2);

                links.Delete(l1);

                links.Unsync.DisposeIfPossible(); // Close links to access log

                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    e.TempTransactionLogFilename);
            }
        }

        [Fact]
        public static void TransactionAutoRevertedTest()
        {
            // Auto Reverted (Because no commit at transaction)
            using (var scope = new TempLinksTestScope(useLog: true))
```

```csharp
            {
                var links = scope.Links;
                var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
                using (var transaction = transactionsLayer.BeginTransaction())
                {
                    var l1 = links.Create();
                    var l2 = links.Create();

                    links.Update(l2, l2, l1, l2);
                }

                Assert.Equal(0UL, links.Count());

                links.Unsync.DisposeIfPossible();

                var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
                ↪   cope.TempTransactionLogFilename);
                Assert.Single(transitions);
            }
        }

        [Fact]
        public static void TransactionUserCodeErrorNoDataSavedTest()
        {
            // User Code Error (Autoreverted), no data saved
            var itself = _constants.Itself;

            TempLinksTestScope lastScope = null;
            try
            {
                using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                ↪   useLog: true))
                {
                    var links = scope.Links;
                    var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
                    ↪   atorBase<ulong>)links.Unsync).Links;
                    using (var transaction = transactionsLayer.BeginTransaction())
                    {
                        var l1 = links.CreateAndUpdate(itself, itself);
                        var l2 = links.CreateAndUpdate(itself, itself);

                        l2 = links.Update(l2, l2, l1, l2);

                        links.CreateAndUpdate(l2, itself);
                        links.CreateAndUpdate(l2, itself);

                        //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
                        ↪   tion>(scope.TempTransactionLogFilename);

                        l2 = links.Update(l2, l1);

                        links.Delete(l2);

                        ExceptionThrower();

                        transaction.Commit();
                    }

                    Global.Trash = links.Count();
                }
            }
            catch
            {
                Assert.False(lastScope == null);

                var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
                ↪   astScope.TempTransactionLogFilename);

                Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
                ↪   transitions[0].After.IsNull());

                lastScope.DeleteFiles();
            }
        }

        [Fact]
        public static void TransactionUserCodeErrorSomeDataSavedTest()
        {
            // User Code Error (Autoreverted), some data saved
```

```csharp
                var itself = _constants.Itself;

            TempLinksTestScope lastScope = null;
            try
            {
                ulong l1;
                ulong l2;

                using (var scope = new TempLinksTestScope(useLog: true))
                {
                    var links = scope.Links;
                    l1 = links.CreateAndUpdate(itself, itself);
                    l2 = links.CreateAndUpdate(itself, itself);

                    l2 = links.Update(l2, l2, l1, l2);

                    links.CreateAndUpdate(l2, itself);
                    links.CreateAndUpdate(l2, itself);

                    links.Unsync.DisposeIfPossible();

                    Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
                    ↪  scope.TempTransactionLogFilename);
                }

                using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                ↪  useLog: true))
                {
                    var links = scope.Links;
                    var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
                    using (var transaction = transactionsLayer.BeginTransaction())
                    {
                        l2 = links.Update(l2, l1);

                        links.Delete(l2);

                        ExceptionThrower();

                        transaction.Commit();
                    }

                    Global.Trash = links.Count();
                }
            }
            catch
            {
                Assert.False(lastScope == null);

                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
                ↪  Scope.TempTransactionLogFilename);

                lastScope.DeleteFiles();
            }
        }

        [Fact]
        public static void TransactionCommit()
        {
            var itself = _constants.Itself;

            var tempDatabaseFilename = Path.GetTempFileName();
            var tempTransactionLogFilename = Path.GetTempFileName();

            // Commit
            using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↪  UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
            using (var links = new UInt64Links(memoryAdapter))
            {
                using (var transaction = memoryAdapter.BeginTransaction())
                {
                    var l1 = links.CreateAndUpdate(itself, itself);
                    var l2 = links.CreateAndUpdate(itself, itself);

                    Global.Trash = links.Update(l2, l2, l1, l2);

                    links.Delete(l1);

                    transaction.Commit();
                }
```

```csharp
                        Global.Trash = links.Count();
                    }

                    Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                    ↪    sactionLogFilename);
                }

        [Fact]
        public static void TransactionDamage()
        {
            var itself = _constants.Itself;

            var tempDatabaseFilename = Path.GetTempFileName();
            var tempTransactionLogFilename = Path.GetTempFileName();

            // Commit
            using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↪    UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
            using (var links = new UInt64Links(memoryAdapter))
            {
                using (var transaction = memoryAdapter.BeginTransaction())
                {
                    var l1 = links.CreateAndUpdate(itself, itself);
                    var l2 = links.CreateAndUpdate(itself, itself);

                    Global.Trash = links.Update(l2, l2, l1, l2);

                    links.Delete(l1);

                    transaction.Commit();
                }

                Global.Trash = links.Count();
            }

            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
            ↪    sactionLogFilename);

            // Damage database

            FileHelpers.WriteFirst(tempTransactionLogFilename, new
            ↪    UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));

            // Try load damaged database
            try
            {
                // TODO: Fix
                using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪    UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
                using (var links = new UInt64Links(memoryAdapter))
                {
                    Global.Trash = links.Count();
                }
            }
            catch (NotSupportedException ex)
            {
                Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
                ↪    yet.");
            }

            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
            ↪    sactionLogFilename);

            File.Delete(tempDatabaseFilename);
            File.Delete(tempTransactionLogFilename);
        }

        [Fact]
        public static void Bug1Test()
        {
            var tempDatabaseFilename = Path.GetTempFileName();
            var tempTransactionLogFilename = Path.GetTempFileName();

            var itself = _constants.Itself;

            // User Code Error (Autoreverted), some data saved
            try
            {
                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             {
336                 using (var transaction = memoryAdapter.BeginTransaction())
337                 {
338                     l2 = links.Update(l2, l1);
339
340                     links.Delete(l2);
341
342                     ExceptionThrower();
343
344                     transaction.Commit();
345                 }
346
347                 Global.Trash = links.Count();
348             }
349         }
350         catch
351         {
352             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp⌋
      ↪  TransactionLogFilename);
353         }
354
355         File.Delete(tempDatabaseFilename);
356         File.Delete(tempTransactionLogFilename);
357     }
358
359     private static void ExceptionThrower() => throw new InvalidOperationException();
360
361     [Fact]
362     public static void PathsTest()
363     {
364         var source = _constants.SourcePart;
365         var target = _constants.TargetPart;
366
367         using (var scope = new TempLinksTestScope())
368         {
369             var links = scope.Links;
370             var l1 = links.CreatePoint();
371             var l2 = links.CreatePoint();
372
373             var r1 = links.GetByKeys(l1, source, target, source);
374             var r2 = links.CheckPathExistance(l2, l2, l2, l2);
375         }
376     }
377
378     [Fact]
379     public static void RecursiveStringFormattingTest()
380     {
381         using (var scope = new TempLinksTestScope(useSequences: true))
382         {
383             var links = scope.Links;
384             var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
385
386             var a = links.CreatePoint();
387             var b = links.CreatePoint();
388             var c = links.CreatePoint();
389
```

```
390                 var ab = links.GetOrCreate(a, b);
391                 var cb = links.GetOrCreate(c, b);
392                 var ac = links.GetOrCreate(a, c);
393
394                 a = links.Update(a, c, b);
395                 b = links.Update(b, a, c);
396                 c = links.Update(c, a, b);
397
398                 Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
399                 Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
400                 Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
401
402                 Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
    ↪    "(5:(4:5 (6:5 4)) 6)");
403                 Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
    ↪    "(6:(5:(4:5 6) 6) 4)");
404                 Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
    ↪    "(4:(5:4 (6:5 4)) 6)");
405
406                 // 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)"
407
408                 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        }
```

```csharp
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
```

```csharp
                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

                ConsoleHelpers.Debug(
                    "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                    ↪  second), counter result: {3}",
                    Iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

        [Fact(Skip = "performance test")]
        public static void TestGetTarget()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
                ↪  Iterations);

                ulong counter = 0;

                //var firstLink = links.First();
                var firstLink = links.Create();

                var sw = Stopwatch.StartNew();

                for (ulong i = 0; i < Iterations; i++)
                {
                    counter += links.GetTarget(firstLink);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

                ConsoleHelpers.Debug(
                    "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                    ↪  second), counter result: {3}",
                    Iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

        [Fact(Skip = "performance test")]
        public static void TestGetTargetInParallel()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
                ↪  parallel.", Iterations);

                long counter = 0;

                //var firstLink = links.First();
                var firstLink = links.Create();

                var sw = Stopwatch.StartNew();

                Parallel.For(0, Iterations, x =>
                {
                    Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
                    //Interlocked.Increment(ref counter);
                });

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

                ConsoleHelpers.Debug(
                    "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                    ↪  second), counter result: {3}",
                    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                {
707                    var source =
↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
708                    var target =
↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
709
710                    counter += links.Search(source, target);
711                }
712
713                var elapsedTime = sw.Elapsed;
714
715                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
716
717                ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪  Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
↪  counter);
718            }
719
720            File.Delete(tempFilename);
721        }*/
722
723        [Fact(Skip = "useless: O(0), was dependent on creation tests")]
724        public static void TestRandomSearchAll()
725        {
726            using (var scope = new TempLinksTestScope())
727            {
728                var links = scope.Links;
729                ulong counter = 0;
730
731                var maxLink = links.Count();
732
733                var iterations = links.Count();
734
735                ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
↪  links.Count());
736
737                var sw = Stopwatch.StartNew();
738
739                for (var i = iterations; i > 0; i--)
740                {
741                    var linksAddressRange = new
↪  Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
742
743                    var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
744                    var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
745
746                    counter += links.SearchOrDefault(source, target);
747                }
748
749                var elapsedTime = sw.Elapsed;
750
751                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
752
753                ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪  Iterations per second), c: {3}",
754                    iterations, elapsedTime, (long)iterationsPerSecond, counter);
```

```csharp
                }
            }

            [Fact(Skip = "useless: O(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);
                }
            }

            /*
            [Fact]
            public static void TestForeach()
            {
                var tempFilename = Path.GetTempFileName();

                using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
            ↪  DefaultLinksSizeStep))
                {
                    ulong counter = 0;

                    ConsoleHelpers.Debug("Testing foreach through links.");

                    var sw = Stopwatch.StartNew();

                    //foreach (var link in links)
                    //{
                    //    counter++;
                    //}

                    var elapsedTime = sw.Elapsed;

                    var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;

                    ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
            ↪  links per second)", counter, elapsedTime, (long)linksPerSecond);
                }

                File.Delete(tempFilename);
            }
            */

            /*
            [Fact]
            public static void TestParallelForeach()
            {
                var tempFilename = Path.GetTempFileName();

                using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
            ↪  DefaultLinksSizeStep))
                {

                    long counter = 0;

                    ConsoleHelpers.Debug("Testing parallel foreach through links.");

                    var sw = Stopwatch.StartNew();

                    //Parallel.ForEach((IEnumerable<ulong>)links, x =>
                    //{
                    //    Interlocked.Increment(ref counter);
                    //});
```

```
831              var elapsedTime = sw.Elapsed;
832
833              var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
834
835              ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
836    ↪   {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: O(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 }
```

1.189 ./csharp/Platform.Data.Doublets.Tests/Uint64LinksExtensionsTests.cs

```
1  using Platform.Data.Doublets.Memory;
2  using Platform.Data.Doublets.Memory.United.Generic;
3  using Platform.Data.Numbers.Raw;
4  using Platform.Memory;
5  using Platform.Numbers;
6  using Xunit;
7  using Xunit.Abstractions;
8  using TLink = System.UInt64;
9
10 namespace Platform.Data.Doublets.Tests
11 {
12     public class Uint64LinksExtensionsTests
13     {
14         public static ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new
           ↪ Platform.IO.TemporaryFile());
15
16         public static ILinks<TLink> CreateLinks<TLink>(string dataDBFilename)
17         {
18             var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
               ↪ true);
19             return new UnitedMemoryLinks<TLink>(new
               ↪ FileMappedResizableDirectMemory(dataDBFilename),
               ↪ UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
               ↪ IndexTreeType.Default);
20         }
21         [Fact]
22         public void FormatStructureWithExternalReferenceTest()
23         {
24             ILinks<TLink> links = CreateLinks();
25             TLink zero = default;
26             var one = Arithmetic.Increment(zero);
27             var markerIndex = one;
28             var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
29             var numberMarker = links.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
               ↪ markerIndex));
30             AddressToRawNumberConverter<TLink> addressToNumberConverter = new();
31             var numberAddress = addressToNumberConverter.Convert(1);
32             var numberLink = links.GetOrCreate(numberMarker, numberAddress);
33             var linkNotation = links.FormatStructure(numberLink, link => link.IsFullPoint(),
               ↪ true);
34             Assert.Equal("(3:(2:1 2) 18446744073709551615)", linkNotation);
35         }
36     }
37 }
```

1.190 ./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  }
```

1.191 ./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();
```

```csharp
                        var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
                        var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
                        TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                        ↪   addressToRawNumberConverter, rawNumberToAddressConverter);
                    }
                }

        private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
        ↪   meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
        ↪   numberToAddressConverter)
        {
            var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
            var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
            ↪   addressToNumberConverter, unicodeSymbolMarker);
            var originalCharacter = 'H';
            var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
            var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
            ↪   unicodeSymbolMarker);
            var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
            ↪   numberToAddressConverter, unicodeSymbolCriterionMatcher);
            var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
            Assert.Equal(originalCharacter, resultingCharacter);
        }

        [Fact]
        public static void StringAndUnicodeSequenceConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;

                var itself = links.Constants.Itself;

                var meaningRoot = links.CreatePoint();
                var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
                var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
                var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
                var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
                var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);

                var powerOf2ToUnaryNumberConverter = new
                ↪   PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
                var addressToUnaryNumberConverter = new
                ↪   AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var charToUnicodeSymbolConverter = new
                ↪   CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                ↪   unicodeSymbolMarker);

                var unaryNumberToAddressConverter = new
                ↪   UnaryNumberToAddressOrOperationConverter<ulong>(links,
                ↪   powerOf2ToUnaryNumberConverter);
                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 stringToUnicodeSequenceConverter = new
                ↪   StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
                ↪   index, optimalVariantConverter, unicodeSequenceMarker);

                var originalString = "Hello";

                var unicodeSequenceLink =
                ↪   stringToUnicodeSequenceConverter.Convert(originalString);

                var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
                ↪   unicodeSymbolMarker);
```

```
97              var unicodeSymbolToCharConverter = new
    ↪      UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
    ↪      unicodeSymbolCriterionMatcher);
98
99              var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
    ↪      unicodeSequenceMarker);
100
101             var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
    ↪      unicodeSymbolCriterionMatcher.IsMatched);
102
103             var unicodeSequenceToStringConverter = new
    ↪      UnicodeSequenceToStringConverter<ulong>(links,
    ↪      unicodeSequenceCriterionMatcher, sequenceWalker,
    ↪      unicodeSymbolToCharConverter);
104
105             var resultingString =
    ↪      unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
106
107             Assert.Equal(originalString, resultingString);
108         }
109     }
110   }
111 }
```

1.192 ./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt32LinksTests.cs

```
1  using System;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Memory;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.Memory.United.Specific;
7  using TLink = System.UInt32;
8
9  namespace Platform.Data.Doublets.Tests
10 {
11     public unsafe static class UnitedMemoryUInt32LinksTests
12     {
13         [Fact]
14         public static void CRUDTest()
15         {
16             Using(links => links.TestCRUDOperations());
17         }
18
19         [Fact]
20         public static void RawNumbersCRUDTest()
21         {
22             Using(links => links.TestRawNumbersCRUDOperations());
23         }
24
25         [Fact]
26         public static void MultipleRandomCreationsAndDeletionsTest()
27         {
28             Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip⏎
    ↪      leRandomCreationsAndDeletions(100));
29         }
30
31         private static void Using(Action<ILinks<TLink>> action)
32         {
33             using (var scope = new Scope<Types<HeapResizableDirectMemory,
    ↪      UInt32UnitedMemoryLinks>>())
34             {
35                 action(scope.Use<ILinks<TLink>>());
36             }
37         }
38     }
39 }
```

1.193 ./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt64LinksTests.cs

```
1  using System;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Memory;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.Memory.United.Specific;
7  using TLink = System.UInt64;
8
9  namespace Platform.Data.Doublets.Tests
10 {
11     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