

## LinksPlatform's Platform.Data.Doublets Class Library

### 1.1 ./csharp/Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.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.Decorators
6 {
7     public class LinksCascadeUniquenessAndUsagesResolver<TLink> : LinksUniquenessResolver<TLink>
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLink> links) : base(links) { }
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
14             ↪ newLinkAddress)
15         {
16             // Use Facade (the last decorator) to ensure recursion working correctly
17             _facade.MergeUsages(oldLinkAddress, newLinkAddress);
18             return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
19         }
20     }
```

### 1.2 ./csharp/Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     /// <remarks>
9     /// <para>Must be used in conjunction with NonNullContentsLinkDeletionResolver.</para>
10     /// <para>Должен использоваться вместе с NonNullContentsLinkDeletionResolver.</para>
11     /// </remarks>
12     public class LinksCascadeUsagesResolver<TLink> : LinksDecoratorBase<TLink>
13     {
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public LinksCascadeUsagesResolver(ILinks<TLink> links) : base(links) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public override void Delete(IList<TLink> restrictions)
19         {
20             var linkIndex = restrictions[_constants.IndexPart];
21             // Use Facade (the last decorator) to ensure recursion working correctly
22             _facade.DeleteAllUsages(linkIndex);
23             _links.Delete(linkIndex);
24         }
25     }
26 }
```

### 1.3 ./csharp/Platform.Data.Doublets/Decorators/LinksDecoratorBase.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     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             }
```

```

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 }

```

#### 1.4 ./csharp/Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Disposables;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5 #pragma warning disable CA1063 // Implement IDisposable Correctly
6
7 namespace Platform.Data.Doublets.Decorators
8 {
9     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
18             {
19                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
20                 get => true;
21             }
22         }
23
24         protected readonly DisposableWithMultipleCallsAllowed Disposable;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected LinksDisposableDecoratorBase(ILinks<TLink> links) : base(links) => Disposable
28             = new DisposableWithMultipleCallsAllowed(Dispose);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         ~LinksDisposableDecoratorBase() => Disposable.Destruct();
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public void Dispose() => Disposable.Dispose();
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected virtual void Dispose(bool manual, bool wasDisposed)
38         {
39             if (!wasDisposed)
40             {
41                 _links.DisposeIfPossible();
42             }
43         }
44     }
45 }

```

## 1.5 ./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
10     ↪ be external (hybrid link's raw number).
11     public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
18         {
19             var links = _links;
20             links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
21             return links.Each(handler, restrictions);
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
26         {
27             // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
28             var links = _links;
29             links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
30             links.EnsureInnerReferenceExists(substitution, nameof(substitution));
31             return links.Update(restrictions, substitution);
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public override void Delete(IList<TLink> restrictions)
36         {
37             var link = restrictions[_constants.IndexPart];
38             var links = _links;
39             links.EnsureLinkExists(link, nameof(link));
40             links.Delete(link);
41         }
42     }

```

## 1.6 ./csharp/Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.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      public class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12         ↪ EqualityComparer<TLink>.Default;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
19         {
20             var constants = _constants;
21             var itselfConstant = constants.Itself;
22             if (!_equalityComparer.Equals(constants.Any, itselfConstant) &&
23             ↪ restrictions.Contains(itselfConstant))
24             {
25                 // Itself constant is not supported for Each method right now, skipping execution
26                 return constants.Continue;
27             }
28             return _links.Each(handler, restrictions);
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
33         ↪ _links.Update(restrictions, _links.ResolveConstantAsSelfReference(_constants.Itself,
34         ↪ restrictions, substitution));

```

```

31     }
32 }

```

### 1.7 ./csharp/Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     /// <remarks>
9     /// Not practical if newSource and newTarget are too big.
10    /// To be able to use practical version we should allow to create link at any specific
11    /// ↪ location inside ResizableDirectMemoryLinks.
12    /// This in turn will require to implement not a list of empty links, but a list of ranges
13    /// ↪ to store it more efficiently.
14    /// </remarks>
15    public class LinksNonExistentDependenciesCreator<TLink> : LinksDecoratorBase<TLink>
16    {
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]
18        public LinksNonExistentDependenciesCreator(ILinks<TLink> links) : base(links) { }
19
20        [MethodImpl(MethodImplOptions.AggressiveInlining)]
21        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
22        {
23            var constants = _constants;
24            var links = _links;
25            links.EnsureCreated(substitution[constants.SourcePart],
26                ↪ substitution[constants.TargetPart]);
27            return links.Update(restrictions, substitution);
28        }
29    }
30 }

```

### 1.8 ./csharp/Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs

```

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

```

### 1.9 ./csharp/Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs

```

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

```

```

20     var newLinkAddress = links.SearchOrDefault(substitution[constants.SourcePart],
21     ↪ substitution[constants.TargetPart]);
22     if (_equalityComparer.Equals(newLinkAddress, default))
23     {
24         return links.Update(restrictions, substitution);
25     }
26     return ResolveAddressChangeConflict(restrictions[constants.IndexPart],
27     ↪ newLinkAddress);
28 }
29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
31 ↪ newLinkAddress)
32 {
33     if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
34     ↪ !_links.Exists(oldLinkAddress))
35     {
36         _facade.Delete(oldLinkAddress);
37     }
38     return newLinkAddress;
39 }
40 }

```

#### 1.10 ./csharp/Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     public class LinksUniquenessValidator<TLink> : LinksDecoratorBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksUniquenessValidator(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
15         {
16             var links = _links;
17             var constants = _constants;
18             links.EnsureDoesNotExists(substitution[constants.SourcePart],
19             ↪ substitution[constants.TargetPart]);
20             return links.Update(restrictions, substitution);
21         }
22     }
23 }

```

#### 1.11 ./csharp/Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     public class 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 }

```

```

29     }
30 }

```

## 1.12 ./csharp/Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     public class 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 }

```

## 1.13 ./csharp/Platform.Data.Doublets/Decorators/UInt64Links.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     /// <summary>
9     /// <para>Represents a combined decorator that implements the basic logic for interacting
10     ↪ with the links storage for links with addresses represented as <see cref="System.UInt64">
11     ↪ </para>
12     /// <para>Представляет комбинированный декоратор, реализующий основную логику по
13     ↪ взаимодействию с хранилищем связей, для связей с адресами представленными в виде <see
14     ↪ cref="System.UInt64"/>.</para>
15     /// </summary>
16     /// <remarks>
17     /// Возможные оптимизации:
18     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
19     /// + меньше объём БД
20     /// - меньше производительность
21     /// - больше ограничение на количество связей в БД)
22     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
23     /// + меньше объём БД
24     /// - больше сложность
25     ///
26     /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
27     ↪ поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
28     ↪ 460 752 303 423 488
29     /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
30     ↪ (битовыми строками) - вариант матрицы (выстраиваемой лениво).
31     ///
32     /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
33     ↪ выбрасываться только при #if DEBUG
34     /// </remarks>
35     public class UInt64Links : LinksDisposableDecoratorBase<ulong>
36     {
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public UInt64Links(ILinks<ulong> links) : base(links) { }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public override ulong Create(IList<ulong> restrictions) => _links.CreatePoint();
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
45         {
46             var constants = _constants;
47             var indexPartConstant = constants.IndexPart;
48             var sourcePartConstant = constants.SourcePart;
49             var targetPartConstant = constants.TargetPart;
50             var nullConstant = constants.Null;
51             var itselfConstant = constants.Itself;

```

```

44     var existedLink = nullConstant;
45     var updatedLink = restrictions[indexPartConstant];
46     var newSource = substitution[sourcePartConstant];
47     var newTarget = substitution[targetPartConstant];
48     var links = _links;
49     if (newSource != itselfConstant && newTarget != itselfConstant)
50     {
51         existedLink = links.SearchOrDefault(newSource, newTarget);
52     }
53     if (existedLink == nullConstant)
54     {
55         var before = links.GetLink(updatedLink);
56         if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
57             ↪ newTarget)
58         {
59             links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
60                 ↪ newSource,
61                                     newTarget == itselfConstant ? updatedLink :
62                 ↪ newTarget);
63         }
64         return updatedLink;
65     }
66     else
67     {
68         return _facade.MergeAndDelete(updatedLink, existedLink);
69     }
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 public override void Delete(ICollection<ulong> restrictions)
74 {
75     var linkIndex = restrictions[_constants.IndexPart];
76     var links = _links;
77     links.EnforceResetValues(linkIndex);
78     _facade.DeleteAllUsages(linkIndex);
79     links.Delete(linkIndex);
80 }
81 }
82 }

```

#### 1.14 ./csharp/Platform.Data.Doublets/Decorators/UniLinks.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Platform.Collections;
5  using Platform.Collections.Lists;
6  using Platform.Data.Universal;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Decorators
11 {
12     /// <remarks>
13     /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
14     /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed
15     ↪ by itself. But can cause creation (update from nothing) or deletion (update to nothing).
16     ///
17     /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
18     ↪ DefaultUniLinksBase, that contains logic itself and can be implemented using both
19     ↪ IDoubletLinks and ILinks.)
20     /// </remarks>
21     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
22     {
23         private static readonly EqualityComparer<TLink> _equalityComparer =
24             ↪ EqualityComparer<TLink>.Default;
25
26         public UniLinks(ILinks<TLink> links) : base(links) { }
27
28         private struct Transition
29         {
30             public IList<TLink> Before;
31             public IList<TLink> After;
32
33             public Transition(IList<TLink> before, IList<TLink> after)
34             {
35                 Before = before;
36                 After = after;
37             }
38         }
39     }
40 }

```

```

36 //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
37 //public static readonly IReadOnlyList<TLink> NullLink = new
    ↳ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
    ↳ });
38
39 // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
    ↳ (Links-Expression)
40 public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
    ↳ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
    ↳ substitutedHandler)
41 {
42     /////List<Transition> transitions = null;
43     /////if (!restriction.IsNullOrEmpty())
44     /////{
45     /////    // Есть причина делать проход (чтение)
46     /////    if (matchedHandler != null)
47     /////    {
48     /////        if (!substitution.IsNullOrEmpty())
49     /////        {
50     /////            // restriction => { 0, 0, 0 } | { 0 } // Create
51     /////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
    ↳ Create / Update
52     /////            // substitution => { 0, 0, 0 } | { 0 } // Delete
53     /////            transitions = new List<Transition>();
54     /////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
55     /////            {
56     /////                // If index is Null, that means we always ignore every other
    ↳ value (they are also Null by definition)
57     /////                var matchDecision = matchedHandler(, NullLink);
58     /////                if (Equals(matchDecision, Constants.Break))
59     /////                {
60     /////                    return false;
61     /////                }
62     /////                if (!Equals(matchDecision, Constants.Skip))
63     /////                {
64     /////                    transitions.Add(new Transition(matchedLink, newValue));
65     /////                }
66     /////            }
67     /////            else
68     /////            {
69     /////                Func<T, bool> handler;
70     /////                handler = link =>
71     /////                {
72     /////                    var matchedLink = Memory.GetLinkValue(link);
73     /////                    var newValue = Memory.GetLinkValue(link);
74     /////                    newValue[Constants.IndexPart] = Constants.Itself;
75     /////                    newValue[Constants.SourcePart] =
    ↳ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
    ↳ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
76     /////                    newValue[Constants.TargetPart] =
    ↳ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
    ↳ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
77     /////                    var matchDecision = matchedHandler(matchedLink, newValue);
78     /////                    if (Equals(matchDecision, Constants.Break))
79     /////                    {
80     /////                        return false;
81     /////                    }
82     /////                    if (!Equals(matchDecision, Constants.Skip))
83     /////                    {
84     /////                        transitions.Add(new Transition(matchedLink, newValue));
85     /////                    }
86     /////                    return true;
87     /////                };
88     /////                if (!Memory.Each(handler, restriction))
89     /////                {
90     /////                    return Constants.Break;
91     /////                }
92     /////            }
93     /////        }
94     /////    }
95     /////    else
96     /////    {
97     /////        Func<T, bool> handler = link =>
98     /////        {
99     /////            var matchedLink = Memory.GetLinkValue(link);
100     /////            var matchDecision = matchedHandler(matchedLink, matchedLink);
101     /////            return !Equals(matchDecision, Constants.Break);
102     /////        };
103     /////        if (!Memory.Each(handler, restriction))
104     /////        {
105     /////            return Constants.Break;
106     /////        }
107     /////    }
108     /////    if (substitution != null)
109     /////    {
110     /////        transitions = new List<Transition>();
111     /////        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 substitutedHandler
142     if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
143         ↪ Constants.Break), restriction))
144         return Constants.Break;
145 //}
146 //else // Update
147 //{
148     //List<IList<T>> matchedLinks = null;
149     if (matchedHandler != null)
150     {
151         matchedLinks = new List<IList<T>>();
152         Func<T, bool> handler = link =>
153         {
154             var matchedLink = Memory.GetLinkValue(link);
155             var matchDecision = matchedHandler(matchedLink);
156             if (Equals(matchDecision, Constants.Break))
157                 return false;
158             if (!Equals(matchDecision, Constants.Skip))
159                 matchedLinks.Add(matchedLink);
160             return true;
161         };
162         if (!Memory.Each(handler, restriction))
163             return Constants.Break;
164     }
165     if (!matchedLinks.IsNullOrEmpty())
166     {
167         var totalMatchedLinks = matchedLinks.Count;
168         for (var i = 0; i < totalMatchedLinks; i++)
169         {
170             var matchedLink = matchedLinks[i];
171             if (substitutedHandler != null)
172             {
173                 var newValue = new List<T>(); // TODO: Prepare value to update here
174                 // TODO: Decide is it actually needed to use Before and After
175                 ↪ substitution handling.
176                 var substitutedDecision = substitutedHandler(matchedLink,
177                 ↪ newValue);
178                 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
185         //             ↪ constant or Skip is enough?
186         //             }
187         //         }
188         //     }
189     //}
190     return _constants.Continue;
191 }
192
193 public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
194 ↪ matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
195 ↪ substitutionHandler)
196 {
197     var constants = _constants;
198     if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
199     {
200         return constants.Continue;
201     }
202     else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
203     ↪ Check if it is a correct condition
204     {
205         // Or it only applies to trigger without matchHandler.
206         throw new NotImplementedException();
207     }
208     else if (!substitution.IsNullOrEmpty()) // Creation
209     {
210         var before = Array.Empty<TLink>();
211         // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
212         ↪ (пройти мимо) или пустить (взять)?
213         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
214         ↪ constants.Break))
215         {
216             return constants.Break;
217         }
218         var after = (IList<TLink>)substitution.ToArray();
219         if (_equalityComparer.Equals(after[0], default))
220         {
221             var newLink = _links.Create();
222             after[0] = newLink;
223         }
224         if (substitution.Count == 1)
225         {
226             after = _links.GetLink(substitution[0]);
227         }
228         else if (substitution.Count == 3)
229         {
230             //Links.Create(after);
231         }
232         else
233         {
234             throw new NotSupportedException();
235         }
236         if (matchHandler != null)
237         {
238             return substitutionHandler(before, after);
239         }
240         return constants.Continue;
241     }
242     else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
243     {
244         if (patternOrCondition.Count == 1)
245         {
246             var linkToDelete = patternOrCondition[0];
247             var before = _links.GetLink(linkToDelete);
248             if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
249             ↪ constants.Break))
250             {
251                 return constants.Break;
252             }
253         }
254     }

```

```

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

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

```

```

323         changes.Add(change);
324         return @continue;
325     });
326     return changes;
327 }
328
329     private TLink AlwaysContinue(ICollection<TLink> linkToMatch) => _constants.Continue;
330 }
331 }

```

### 1.15 ./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      public struct Doublet<T> : IEquatable<Doublet<T>>
10     {
11         private static readonly EqualityComparer<T> _equalityComparer =
12             EqualityComparer<T>.Default;
13
14         public T Source
15         {
16             [MethodImpl(MethodImplOptions.AggressiveInlining)]
17             get;
18             [MethodImpl(MethodImplOptions.AggressiveInlining)]
19             set;
20         }
21         public T Target
22         {
23             [MethodImpl(MethodImplOptions.AggressiveInlining)]
24             get;
25             [MethodImpl(MethodImplOptions.AggressiveInlining)]
26             set;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public Doublet(T source, T target)
31         {
32             Source = source;
33             Target = target;
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public override string ToString() => $"{Source}->{Target}";
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
41             && _equalityComparer.Equals(Target, other.Target);
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public override bool Equals(object obj) => obj is Doublet<T> doublet ?
45             base.Equals(doublet) : false;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         public override int GetHashCode() => (Source, Target).GetHashCode();
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public static bool operator !=(Doublet<T> left, Doublet<T> right) => !(left == right);
55     }
56 }

```

### 1.16 ./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 }

```

### 1.17 ./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 }

```

### 1.18 ./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;
7 using Platform.Collections.Arrays;
8 using Platform.Random;
9 using Platform.Setters;
10 using Platform.Converters;
11 using Platform.Numbers;
12 using Platform.Data.Exceptions;
13 using Platform.Data.Doublets.Decorators;
14
15 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17 namespace Platform.Data.Doublets
18 {
19     public static class ILinksExtensions
20     {
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public static void RunRandomCreations<TLink>(this ILinks<TLink> links, ulong
23             ↪ amountOfCreations)
24         {
25             var random = RandomHelpers.Default;
26             var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
27             var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
28             for (var i = 0UL; i < amountOfCreations; i++)
29             {
30                 var linksAddressRange = new Range<ulong>(0,
31                     ↪ addressToUInt64Converter.Convert(links.Count()));
32                 var source =
33                     ↪ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
34                 var target =
35                     ↪ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
36                 links.GetOrCreate(source, target);
37             }
38         }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public static void RunRandomSearches<TLink>(this ILinks<TLink> links, ulong
42             ↪ amountOfSearches)
43         {
44             var random = RandomHelpers.Default;
45             var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
46             var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
47             for (var i = 0UL; i < amountOfSearches; i++)
48             {
49                 var linksAddressRange = new Range<ulong>(0,
50                     ↪ addressToUInt64Converter.Convert(links.Count()));
51                 var source =
52                     ↪ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
53                 var target =
54                     ↪ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
55                 links.SearchOrDefault(source, target);
56             }
57         }
58     }
59 }

```

```

48     }
49 }
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, ulong
↳ amountOfDeletions)
53 {
54     var random = RandomHelpers.Default;
55     var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
56     var uint64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
57     var linksCount = addressToUInt64Converter.Convert(links.Count());
58     var min = amountOfDeletions > linksCount ? OUL : linksCount - amountOfDeletions;
59     for (var i = OUL; i < amountOfDeletions; i++)
60     {
61         linksCount = addressToUInt64Converter.Convert(links.Count());
62         if (linksCount <= min)
63         {
64             break;
65         }
66         var linksAddressRange = new Range<ulong>(min, linksCount);
67         var link =
↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
68         links.Delete(link);
69     }
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
↳ links.Delete(new LinkAddress<TLink>(linkToDelete));
74
75 /// <remarks>
76 /// TODO: Возможно есть очень простой способ это сделать.
77 /// (Например просто удалить файл, или изменить его размер таким образом,
78 /// чтобы удалился весь контент)
79 /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
80 /// </remarks>
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 public static void DeleteAll<TLink>(this ILinks<TLink> links)
83 {
84     var equalityComparer = EqualityComparer<TLink>.Default;
85     var comparer = Comparer<TLink>.Default;
86     for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
↳ Arithmetic.Decrement(i))
87     {
88         links.Delete(i);
89         if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
90         {
91             i = links.Count();
92         }
93     }
94 }
95
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 public static TLink First<TLink>(this ILinks<TLink> links)
98 {
99     TLink firstLink = default;
100     var equalityComparer = EqualityComparer<TLink>.Default;
101     if (equalityComparer.Equals(links.Count(), default))
102     {
103         throw new InvalidOperationException("В хранилище нет связей.");
104     }
105     links.Each(links.Constants.Any, links.Constants.Any, link =>
106     {
107         firstLink = link[links.Constants.IndexPart];
108         return links.Constants.Break;
109     });
110     if (equalityComparer.Equals(firstLink, default))
111     {
112         throw new InvalidOperationException("В процессе поиска по хранилищу не было
↳ найдено связей.");
113     }
114     return firstLink;
115 }
116
117 #region Paths
118
119 /// <remarks>
120 /// TODO: Как так? Как то что ниже может быть корректно?
121 /// Скорее всего практически не применимо

```

```

122  /// Предполагалось, что можно было конвертировать формируемый в проходе через
123  ↪ SequenceWalker
124  /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
125  /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
126  /// </remarks>
127  [MethodImpl(MethodImplOptions.AggressiveInlining)]
128  public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
129  ↪ path)
130  {
131      var current = path[0];
132      //EnsureLinkExists(current, "path");
133      if (!links.Exists(current))
134      {
135          return false;
136      }
137      var equalityComparer = EqualityComparer<TLink>.Default;
138      var constants = links.Constants;
139      for (var i = 1; i < path.Length; i++)
140      {
141          var next = path[i];
142          var values = links.GetLink(current);
143          var source = values[constants.SourcePart];
144          var target = values[constants.TargetPart];
145          if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
146          ↪ next))
147          {
148              //throw new InvalidOperationException(string.Format("Невозможно выбрать
149              ↪ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
150              return false;
151          }
152          if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
153          ↪ target))
154          {
155              //throw new InvalidOperationException(string.Format("Невозможно продолжить
156              ↪ путь через элемент пути {0}", next));
157              return false;
158          }
159          current = next;
160      }
161      return true;
162  }
163
164  /// <remarks>
165  /// Может потребовать дополнительного стека для PathElement's при использовании
166  ↪ SequenceWalker.
167  /// </remarks>
168  [MethodImpl(MethodImplOptions.AggressiveInlining)]
169  public static TLink GetByKeyes<TLink>(this ILinks<TLink> links, TLink root, params int[]
170  ↪ path)
171  {
172      links.EnsureLinkExists(root, "root");
173      var currentLink = root;
174      for (var i = 0; i < path.Length; i++)
175      {
176          currentLink = links.GetLink(currentLink)[path[i]];
177      }
178      return currentLink;
179  }
180
181  [MethodImpl(MethodImplOptions.AggressiveInlining)]
182  public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
183  ↪ links, TLink root, ulong size, ulong index)
184  {
185      var constants = links.Constants;
186      var source = constants.SourcePart;
187      var target = constants.TargetPart;
188      if (!Platform.Numbers.Math.IsPowerOfTwo(size))
189      {
190          throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
191          ↪ than powers of two are not supported.");
192      }
193      var path = new BitArray(BitConverter.GetBytes(index));
194      var length = Bit.GetLowestPosition(size);
195      links.EnsureLinkExists(root, "root");
196      var currentLink = root;
197      for (var i = length - 1; i >= 0; i--)
198      {
199          currentLink = links.GetLink(currentLink)[path[i] ? target : source];
200      }
201  }

```

```

190     }
191     return currentLink;
192 }
193
194 #endregion
195
196 /// <summary>
197 /// Возвращает индекс указанной связи.
198 /// </summary>
199 /// <param name="links">Хранилище связей.</param>
200 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    → содержимого.</param>
201 /// <returns>Индекс начальной связи для указанной связи.</returns>
202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    → link[links.Constants.IndexPart];
204
205 /// <summary>
206 /// Возвращает индекс начальной (Source) связи для указанной связи.
207 /// </summary>
208 /// <param name="links">Хранилище связей.</param>
209 /// <param name="link">Индекс связи.</param>
210 /// <returns>Индекс начальной связи для указанной связи.</returns>
211 [MethodImpl(MethodImplOptions.AggressiveInlining)]
212 public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
    → links.GetLink(link)[links.Constants.SourcePart];
213
214 /// <summary>
215 /// Возвращает индекс начальной (Source) связи для указанной связи.
216 /// </summary>
217 /// <param name="links">Хранилище связей.</param>
218 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    → содержимого.</param>
219 /// <returns>Индекс начальной связи для указанной связи.</returns>
220 [MethodImpl(MethodImplOptions.AggressiveInlining)]
221 public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    → link[links.Constants.SourcePart];
222
223 /// <summary>
224 /// Возвращает индекс конечной (Target) связи для указанной связи.
225 /// </summary>
226 /// <param name="links">Хранилище связей.</param>
227 /// <param name="link">Индекс связи.</param>
228 /// <returns>Индекс конечной связи для указанной связи.</returns>
229 [MethodImpl(MethodImplOptions.AggressiveInlining)]
230 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
    → links.GetLink(link)[links.Constants.TargetPart];
231
232 /// <summary>
233 /// Возвращает индекс конечной (Target) связи для указанной связи.
234 /// </summary>
235 /// <param name="links">Хранилище связей.</param>
236 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    → содержимого.</param>
237 /// <returns>Индекс конечной связи для указанной связи.</returns>
238 [MethodImpl(MethodImplOptions.AggressiveInlining)]
239 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    → link[links.Constants.TargetPart];
240
241 /// <summary>
242 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    → (handler) для каждой подходящей связи.
243 /// </summary>
244 /// <param name="links">Хранилище связей.</param>
245 /// <param name="handler">Обработчик каждой подходящей связи.</param>
246 /// <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>
247 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    → случае.</returns>
248 [MethodImpl(MethodImplOptions.AggressiveInlining)]
249 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
    → handler, params TLink[] restrictions)
    => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
    → links.Constants.Continue);
250
251 /// <summary>
252

```



```

253 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
254   ↳ (handler) для каждой подходящей связи.
255 /// </summary>
256 /// <param name="links">Хранилище связей.</param>
257 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
258   ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
259   ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
260 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
261   ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
262   ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
263 /// <param name="handler">Обработчик каждой подходящей связи.</param>
264 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
265   ↳ случае.</returns>
266 [MethodImpl(MethodImplOptions.AggressiveInlining)]
267 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
268   ↳ Func<TLink, bool> handler)
269 {
270     var constants = links.Constants;
271     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
272       ↳ constants.Break, constants.Any, source, target);
273 }
274
275 /// <summary>
276 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
277   ↳ (handler) для каждой подходящей связи.
278 /// </summary>
279 /// <param name="links">Хранилище связей.</param>
280 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
281   ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
282   ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
283 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
284   ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
285   ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
286 /// <param name="handler">Обработчик каждой подходящей связи.</param>
287 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
288   ↳ случае.</returns>
289 [MethodImpl(MethodImplOptions.AggressiveInlining)]
290 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
291   ↳ Func<IList<TLink>, TLink> handler) => links.Each(handler, links.Constants.Any,
292   ↳ source, target);
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
296   ↳ restrictions)
297 {
298     var arraySize = CheckedConverter<TLink,
299       ↳ long>.Default.Convert(links.Count(restrictions));
300     if (arraySize > 0)
301     {
302         var array = new IList<TLink>[arraySize];
303         var filler = new ArrayFiller<IList<TLink>, TLink>(array,
304           ↳ links.Constants.Continue);
305         links.Each(filler.AddAndReturnConstant, restrictions);
306         return array;
307     }
308     else
309     {
310         return Array.Empty<IList<TLink>>();
311     }
312 }
313
314 [MethodImpl(MethodImplOptions.AggressiveInlining)]
315 public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
316   ↳ restrictions)
317 {
318     var arraySize = CheckedConverter<TLink,
319       ↳ long>.Default.Convert(links.Count(restrictions));
320     if (arraySize > 0)
321     {
322         var array = new TLink[arraySize];
323         var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
324         links.Each(filler.AddFirstAndReturnConstant, restrictions);
325         return array;
326     }
327     else
328     {
329         return Array.Empty<TLink>();
330     }
331 }

```

```

309     }
310 }
311
312 /// <summary>
313 /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
314   ↳ в хранилище связей.
315 /// </summary>
316 /// <param name="links">Хранилище связей.</param>
317 /// <param name="source">Начало связи.</param>
318 /// <param name="target">Конец связи.</param>
319 /// <returns>Значение, определяющее существует ли связь.</returns>
320 [MethodImpl(MethodImplOptions.AggressiveInlining)]
321 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
322   ↳ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
323   ↳ default) > 0;
324
325 #region Ensure
326 // TODO: May be move to EnsureExtensions or make it both there and here
327
328 [MethodImpl(MethodImplOptions.AggressiveInlining)]
329 public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
330   ↳ restrictions)
331 {
332     for (var i = 0; i < restrictions.Count; i++)
333     {
334         if (!links.Exists(restrictions[i]))
335         {
336             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
337               ↳ $"sequence[{i}]");
338         }
339     }
340 }
341
342 [MethodImpl(MethodImplOptions.AggressiveInlining)]
343 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
344   ↳ reference, string argumentName)
345 {
346     if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
347     {
348         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
349     }
350 }
351
352 [MethodImpl(MethodImplOptions.AggressiveInlining)]
353 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
354   ↳ IList<TLink> restrictions, string argumentName)
355 {
356     for (int i = 0; i < restrictions.Count; i++)
357     {
358         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
359     }
360 }
361
362 [MethodImpl(MethodImplOptions.AggressiveInlining)]
363 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
364   ↳ restrictions)
365 {
366     var equalityComparer = EqualityComparer<TLink>.Default;
367     var any = links.Constants.Any;
368     for (var i = 0; i < restrictions.Count; i++)
369     {
370         if (!equalityComparer.Equals(restrictions[i], any) &&
371           ↳ !links.Exists(restrictions[i]))
372         {
373             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
374               ↳ $"sequence[{i}]");
375         }
376     }
377 }
378
379 [MethodImpl(MethodImplOptions.AggressiveInlining)]
380 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
381   ↳ string argumentName)
382 {
383     var equalityComparer = EqualityComparer<TLink>.Default;
384     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
385     {
386

```

```

375         throw new ArgumentException<TLink>(link, argumentName);
376     }
377 }
378
379 [MethodImpl(MethodImplOptions.AggressiveInlining)]
380 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
    ↳ link, string argumentName)
381 {
382     var equalityComparer = EqualityComparer<TLink>.Default;
383     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
384     {
385         throw new ArgumentException<TLink>(link, argumentName);
386     }
387 }
388
389 /// <param name="links">Хранилище связей.</param>
390 [MethodImpl(MethodImplOptions.AggressiveInlining)]
391 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target)
392 {
393     if (links.Exists(source, target))
394     {
395         throw new LinkWithSameValueAlreadyExistsException();
396     }
397 }
398
399 /// <param name="links">Хранилище связей.</param>
400 [MethodImpl(MethodImplOptions.AggressiveInlining)]
401 public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
402 {
403     if (links.HasUsages(link))
404     {
405         throw new ArgumentExceptionHasDependenciesException<TLink>(link);
406     }
407 }
408
409 /// <param name="links">Хранилище связей.</param>
410 [MethodImpl(MethodImplOptions.AggressiveInlining)]
411 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.Create, addresses);
412
413 /// <param name="links">Хранилище связей.</param>
414 [MethodImpl(MethodImplOptions.AggressiveInlining)]
415 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
416
417 /// <param name="links">Хранилище связей.</param>
418 [MethodImpl(MethodImplOptions.AggressiveInlining)]
419 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
    ↳ params TLink[] addresses)
420 {
421     var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
422     var uint64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
423     var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
    ↳ !links.Exists(x)));
424     if (nonExistentAddresses.Count > 0)
425     {
426         var max = nonExistentAddresses.Max();
427         max = uint64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.
    ↳ Convert(max),
    ↳ addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Max
    ↳ imum)));
428         var createdLinks = new List<TLink>();
429         var equalityComparer = EqualityComparer<TLink>.Default;
430         TLink createdLink = creator();
431         while (!equalityComparer.Equals(createdLink, max))
432         {
433             createdLinks.Add(createdLink);
434         }
435         for (var i = 0; i < createdLinks.Count; i++)
436         {
437             if (!nonExistentAddresses.Contains(createdLinks[i]))
438             {
439                 links.Delete(createdLinks[i]);
440             }
441         }
442     }
443 }

```

```

444 #endregion
445
446
447 /// <param name="links">Хранилище связей.</param>
448 [MethodImpl(MethodImplOptions.AggressiveInlining)]
449 public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
450 {
451     var constants = links.Constants;
452     var values = links.GetLink(link);
453     TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
454         ↪ constants.Any));
455     var equalityComparer = EqualityComparer<TLink>.Default;
456     if (equalityComparer.Equals(values[constants.SourcePart], link))
457     {
458         usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
459     }
460     TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
461         ↪ link));
462     if (equalityComparer.Equals(values[constants.TargetPart], link))
463     {
464         usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
465     }
466     return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
467 }
468
469 /// <param name="links">Хранилище связей.</param>
470 [MethodImpl(MethodImplOptions.AggressiveInlining)]
471 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
472     ↪ Comparer<TLink>.Default.Compare(links.CountUsages(link), default) > 0;
473
474 /// <param name="links">Хранилище связей.</param>
475 [MethodImpl(MethodImplOptions.AggressiveInlining)]
476 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
477     ↪ TLink target)
478 {
479     var constants = links.Constants;
480     var values = links.GetLink(link);
481     var equalityComparer = EqualityComparer<TLink>.Default;
482     return equalityComparer.Equals(values[constants.SourcePart], source) &&
483         ↪ equalityComparer.Equals(values[constants.TargetPart], target);
484 }
485
486 /// <summary>
487 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
488 /// </summary>
489 /// <param name="links">Хранилище связей.</param>
490 /// <param name="source">Индекс связи, которая является началом для искомой
491     ↪ связи.</param>
492 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
493 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
494     ↪ (концом).</returns>
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
497     ↪ target)
498 {
499     var constants = links.Constants;
500     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
501     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
502     return setter.Result;
503 }
504
505 /// <param name="links">Хранилище связей.</param>
506 [MethodImpl(MethodImplOptions.AggressiveInlining)]
507 public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
508
509 /// <param name="links">Хранилище связей.</param>
510 [MethodImpl(MethodImplOptions.AggressiveInlining)]
511 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
512 {
513     var link = links.Create();
514     return links.Update(link, link, link);
515 }
516
517 /// <param name="links">Хранилище связей.</param>
518 [MethodImpl(MethodImplOptions.AggressiveInlining)]
519 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
520     ↪ target) => links.Update(links.Create(), source, target);

```

```

513 /// <summary>
514 /// Обновляет связь с указанными началом (Source) и концом (Target)
515 /// на связь с указанными началом (NewSource) и концом (NewTarget).
516 /// </summary>
517 /// <param name="links">Хранилище связей.</param>
518 /// <param name="link">Индекс обновляемой связи.</param>
519 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↳ выполняется обновление.</param>
520 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↳ выполняется обновление.</param>
521 /// <returns>Индекс обновлённой связи.</returns>
522 [MethodImpl(MethodImplOptions.AggressiveInlining)]
523 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));

524 /// <summary>
525 /// Обновляет связь с указанными началом (Source) и концом (Target)
526 /// на связь с указанными началом (NewSource) и концом (NewTarget).
527 /// </summary>
528 /// <param name="links">Хранилище связей.</param>
529 /// <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.. $\infty$  конкретный адрес другой
    ↳ связи.</param>
531 /// <returns>Индекс обновлённой связи.</returns>
532 [MethodImpl(MethodImplOptions.AggressiveInlining)]
533 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
534 {
535     if (restrictions.Length == 2)
536     {
537         return links.MergeAndDelete(restrictions[0], restrictions[1]);
538     }
539     if (restrictions.Length == 4)
540     {
541         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
    ↳ restrictions[2], restrictions[3]);
542     }
543     else
544     {
545         return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
546     }
547 }

548 [MethodImpl(MethodImplOptions.AggressiveInlining)]
549 public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
    ↳ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
550 {
551     var equalityComparer = EqualityComparer<TLink>.Default;
552     var constants = links.Constants;
553     var restrictionsIndex = restrictions[constants.IndexPart];
554     var substitutionIndex = substitution[constants.IndexPart];
555     if (equalityComparer.Equals(substitutionIndex, default))
556     {
557         substitutionIndex = restrictionsIndex;
558     }
559     var source = substitution[constants.SourcePart];
560     var target = substitution[constants.TargetPart];
561     source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
562     target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
563     return new Link<TLink>(substitutionIndex, source, target);
564 }

565 /// <summary>
566 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
    ↳ с указанными Source (началом) и Target (концом).
567 /// </summary>
568 /// <param name="links">Хранилище связей.</param>
569 /// <param name="source">Индекс связи, которая является началом на создаваемой
    ↳ связи.</param>
570 /// <param name="target">Индекс связи, которая является концом для создаваемой
    ↳ связи.</param>
571 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
572 [MethodImpl(MethodImplOptions.AggressiveInlining)]
573 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)
574 {
575

```

```

577     var link = links.SearchOrDefault(source, target);
578     if (EqualityComparer<TLink>.Default.Equals(link, default))
579     {
580         link = links.CreateAndUpdate(source, target);
581     }
582     return link;
583 }
584
585 /// <summary>
586 /// Обновляет связь с указанными началом (Source) и концом (Target)
587 /// на связь с указанными началом (NewSource) и концом (NewTarget).
588 /// </summary>
589 /// <param name="links">Хранилище связей.</param>
590 /// <param name="source">Индекс связи, которая является началом обновляемой
    → связи.</param>
591 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
592 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    → выполняется обновление.</param>
593 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    → выполняется обновление.</param>
594 /// <returns>Индекс обновлённой связи.</returns>
595 [MethodImpl(MethodImplOptions.AggressiveInlining)]
596 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
    → TLink target, TLink newSource, TLink newTarget)
597 {
598     var equalityComparer = EqualityComparer<TLink>.Default;
599     var link = links.SearchOrDefault(source, target);
600     if (equalityComparer.Equals(link, default))
601     {
602         return links.CreateAndUpdate(newSource, newTarget);
603     }
604     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
    → target))
605     {
606         return link;
607     }
608     return links.Update(link, newSource, newTarget);
609 }
610
611 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
612 /// <param name="links">Хранилище связей.</param>
613 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
614 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
615 [MethodImpl(MethodImplOptions.AggressiveInlining)]
616 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
    → target)
617 {
618     var link = links.SearchOrDefault(source, target);
619     if (!EqualityComparer<TLink>.Default.Equals(link, default))
620     {
621         links.Delete(link);
622         return link;
623     }
624     return default;
625 }
626
627 /// <summary>Удаляет несколько связей.</summary>
628 /// <param name="links">Хранилище связей.</param>
629 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
630 [MethodImpl(MethodImplOptions.AggressiveInlining)]
631 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
632 {
633     for (int i = 0; i < deletedLinks.Count; i++)
634     {
635         links.Delete(deletedLinks[i]);
636     }
637 }
638
639 /// <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>
640 [MethodImpl(MethodImplOptions.AggressiveInlining)]
641 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
642 {
643     var anyConstant = links.Constants.Any;
644     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
645     links.DeleteByQuery(usagesAsSourceQuery);
646     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);

```

```

647     links.DeleteByQuery(usagesAsTargetQuery);
648 }
649
650 [MethodImpl(MethodImplOptions.AggressiveInlining)]
651 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
652 {
653     var count = CheckedConverter<TLink, long>.Default.Convert(links.Count(query));
654     if (count > 0)
655     {
656         var queryResult = new TLink[count];
657         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
658             ↪ links.Constants.Continue);
659         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
660         for (var i = count - 1; i >= 0; i--)
661         {
662             links.Delete(queryResult[i]);
663         }
664     }
665 }
666
667 // TODO: Move to Platform.Data
668 [MethodImpl(MethodImplOptions.AggressiveInlining)]
669 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
670 {
671     var nullConstant = links.Constants.Null;
672     var equalityComparer = EqualityComparer<TLink>.Default;
673     var link = links.GetLink(linkIndex);
674     for (int i = 1; i < link.Count; i++)
675     {
676         if (!equalityComparer.Equals(link[i], nullConstant))
677         {
678             return false;
679         }
680     }
681     return true;
682 }
683
684 // TODO: Create a universal version of this method in Platform.Data (with using of for
685 ↪ loop)
686 [MethodImpl(MethodImplOptions.AggressiveInlining)]
687 public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
688 {
689     var nullConstant = links.Constants.Null;
690     var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
691     links.Update(updateRequest);
692 }
693
694 // TODO: Create a universal version of this method in Platform.Data (with using of for
695 ↪ loop)
696 [MethodImpl(MethodImplOptions.AggressiveInlining)]
697 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
698 {
699     if (!links.AreValuesReset(linkIndex))
700     {
701         links.ResetValues(linkIndex);
702     }
703 }
704
705 /// <summary>
706 /// Merging two usages graphs, all children of old link moved to be children of new link
707 ↪ or deleted.
708 /// </summary>
709 [MethodImpl(MethodImplOptions.AggressiveInlining)]
710 public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
711 ↪ TLink newLinkIndex)
712 {
713     var addressToInt64Converter = CheckedConverter<TLink, long>.Default;
714     var equalityComparer = EqualityComparer<TLink>.Default;
715     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
716     {
717         var constants = links.Constants;
718         var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
719             ↪ constants.Any);
720         var usagesAsSourceCount =
721             ↪ addressToInt64Converter.Convert(links.Count(usagesAsSourceQuery));
722         var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
723             ↪ oldLinkIndex);

```

```

716     var usagesAsTargetCount =
717         ↪ addressToInt64Converter.Convert(links.Count(usagesAsTargetQuery));
718     var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
719         ↪ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
720     if (!isStandalonePoint)
721     {
722         var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
723         if (totalUsages > 0)
724         {
725             var usages = ArrayPool.Allocate<TLink>(totalUsages);
726             var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
727                 ↪ links.Constants.Continue);
728             var i = 0L;
729             if (usagesAsSourceCount > 0)
730             {
731                 links.Each(usagesFiller.AddFirstAndReturnConstant,
732                     ↪ usagesAsSourceQuery);
733                 for (; i < usagesAsSourceCount; i++)
734                 {
735                     var usage = usages[i];
736                     if (!equalityComparer.Equals(usage, oldLinkIndex))
737                     {
738                         links.Update(usage, newLinkIndex, links.GetTarget(usage));
739                     }
740                 }
741             }
742             if (usagesAsTargetCount > 0)
743             {
744                 links.Each(usagesFiller.AddFirstAndReturnConstant,
745                     ↪ usagesAsTargetQuery);
746                 for (; i < usages.Length; i++)
747                 {
748                     var usage = usages[i];
749                     if (!equalityComparer.Equals(usage, oldLinkIndex))
750                     {
751                         links.Update(usage, links.GetSource(usage), newLinkIndex);
752                     }
753                 }
754             }
755             ArrayPool.Free(usages);
756         }
757     }
758     return newLinkIndex;
759 }
760
761 /// <summary>
762 /// Replace one link with another (replaced link is deleted, children are updated or
763 /// ↪ deleted).
764 /// </summary>
765 [MethodImpl(MethodImplOptions.AggressiveInlining)]
766 public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
767     ↪ TLink newLinkIndex)
768 {
769     var equalityComparer = EqualityComparer<TLink>.Default;
770     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
771     {
772         links.MergeUsages(oldLinkIndex, newLinkIndex);
773         links.Delete(oldLinkIndex);
774     }
775     return newLinkIndex;
776 }
777
778 [MethodImpl(MethodImplOptions.AggressiveInlining)]
779 public static ILinks<TLink>
780     ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
781 {
782     links = new LinksCascadeUsagesResolver<TLink>(links);
783     links = new NonNullContentsLinkDeletionResolver<TLink>(links);
784     links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
785     return links;
786 }
787 }

```

## 1.19 ./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 }

```

## 1.20 ./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 =
12            ↳ EqualityComparer<TLink>.Default;
13
14        private readonly TLink _frequencyMarker;
15        private readonly TLink _unaryOne;
16        private readonly IIncrementer<TLink> _unaryNumberIncrementer;
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
20            ↳ IIncrementer<TLink> unaryNumberIncrementer)
21            : base(links)
22        {
23            _frequencyMarker = frequencyMarker;
24            _unaryOne = unaryOne;
25            _unaryNumberIncrementer = unaryNumberIncrementer;
26        }
27
28        [MethodImpl(MethodImplOptions.AggressiveInlining)]
29        public TLink Increment(TLink frequency)
30        {
31            var links = _links;
32            if (_equalityComparer.Equals(frequency, default))
33            {
34                return links.GetOrCreate(_unaryOne, _frequencyMarker);
35            }
36            var incrementedSource =
37                ↳ _unaryNumberIncrementer.Increment(links.GetSource(frequency));
38            return links.GetOrCreate(incrementedSource, _frequencyMarker);
39        }
40    }
41 }

```

## 1.21 ./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 =
12            ↳ EqualityComparer<TLink>.Default;
13
14        private readonly TLink _unaryOne;
15
16        [MethodImpl(MethodImplOptions.AggressiveInlining)]
17        public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
18            ↳ _unaryOne = unaryOne;
19
20        [MethodImpl(MethodImplOptions.AggressiveInlining)]
21        public TLink Increment(TLink unaryNumber)
22        {
23            var links = _links;
24            if (_equalityComparer.Equals(unaryNumber, _unaryOne))
25            {
26                return links.GetOrCreate(_unaryOne, _unaryOne);
27            }
28            var source = links.GetSource(unaryNumber);
29        }
30    }
31 }

```

```

27     var target = links.GetTarget(unaryNumber);
28     if (_equalityComparer.Equals(source, target))
29     {
30         return links.GetOrCreate(unaryNumber, _unaryOne);
31     }
32     else
33     {
34         return links.GetOrCreate(source, Increment(target));
35     }
36 }
37 }
38 }

```

## 1.22 ./csharp/Platform.Data.Doublets/Link.cs

```

1  using Platform.Collections.Lists;
2  using Platform.Exceptions;
3  using Platform.Ranges;
4  using Platform.Singletons;
5  using System;
6  using System.Collections;
7  using System.Collections.Generic;
8  using System.Runtime.CompilerServices;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets
13 {
14     /// <summary>
15     /// Структура описывающая уникальную связь.
16     /// </summary>
17     public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
18     {
19         public static readonly Link<TLink> Null = new Link<TLink>();
20
21         private static readonly LinksConstants<TLink> _constants =
22             ↪ Default<LinksConstants<TLink>>.Instance;
23         private static readonly EqualityComparer<TLink> _equalityComparer =
24             ↪ EqualityComparer<TLink>.Default;
25
26         private const int Length = 3;
27
28         public readonly TLink Index;
29         public readonly TLink Source;
30         public readonly TLink Target;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
34             ↪ Target);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public Link(IList<TLink> values) => SetValues(values, out Index, out Source, out Target);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public Link(object other)
41         {
42             if (other is Link<TLink> otherLink)
43             {
44                 SetValues(ref otherLink, out Index, out Source, out Target);
45             }
46             else if (other is IList<TLink> otherList)
47             {
48                 SetValues(otherList, out Index, out Source, out Target);
49             }
50             else
51             {
52                 throw new NotSupportedException();
53             }
54         }
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
58             ↪ Target);
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public Link(TLink index, TLink source, TLink target)
62         {
63             Index = index;
64             Source = source;
65             Target = target;
66         }
67     }
68 }

```

```

63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
65     ↪ 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(ICollection<TLink> values, out TLink index, out TLink source,
74     ↪ out TLink target)
75 {
76     switch (values.Count)
77     {
78         case 3:
79             index = values[0];
80             source = values[1];
81             target = values[2];
82             break;
83         case 2:
84             index = values[0];
85             source = values[1];
86             target = default;
87             break;
88         case 1:
89             index = values[0];
90             source = default;
91             target = default;
92             break;
93         default:
94             index = default;
95             source = default;
96             target = default;
97             break;
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> &&
110     ↪ Equals((Link<TLink>)other);
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
114     && _equalityComparer.Equals(Source, other.Source)
115     && _equalityComparer.Equals(Target, other.Target);
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 public static string ToString(TLink index, TLink source, TLink target) => $"{index}:
119     ↪ {source}->{target}";
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 public static string ToString(TLink source, TLink target) => $"{source}->{target}";
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
126
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 public static implicit operator Link<TLink> (TLink[] linkArray) => new
129     ↪ Link<TLink>(linkArray);
130
131 [MethodImpl(MethodImplOptions.AggressiveInlining)]
132 public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
133     ↪ ToString(Source, Target) : ToString(Index, Source, Target);
134
135 #region IList
136 public int Count
137 {
138     [MethodImpl(MethodImplOptions.AggressiveInlining)]
139     get => Length;
140 }

```

```

137     }
138
139     public bool IsReadOnly
140     {
141         [MethodImpl(MethodImplOptions.AggressiveInlining)]
142         get => true;
143     }
144
145     public TLink this[int index]
146     {
147         [MethodImpl(MethodImplOptions.AggressiveInlining)]
148         get
149         {
150             Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
151                 ↪ nameof(index));
152             if (index == _constants.IndexPart)
153             {
154                 return Index;
155             }
156             if (index == _constants.SourcePart)
157             {
158                 return Source;
159             }
160             if (index == _constants.TargetPart)
161             {
162                 return Target;
163             }
164             throw new NotSupportedException(); // Impossible path due to
165                 ↪ Ensure.ArgumentInRange
166         }
167         [MethodImpl(MethodImplOptions.AggressiveInlining)]
168         set => throw new NotSupportedException();
169     }
170
171     [MethodImpl(MethodImplOptions.AggressiveInlining)]
172     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
173
174     [MethodImpl(MethodImplOptions.AggressiveInlining)]
175     public IEnumerator<TLink> GetEnumerator()
176     {
177         yield return Index;
178         yield return Source;
179         yield return Target;
180     }
181
182     [MethodImpl(MethodImplOptions.AggressiveInlining)]
183     public void Add(TLink item) => throw new NotSupportedException();
184
185     [MethodImpl(MethodImplOptions.AggressiveInlining)]
186     public void Clear() => throw new NotSupportedException();
187
188     [MethodImpl(MethodImplOptions.AggressiveInlining)]
189     public bool Contains(TLink item) => IndexOf(item) >= 0;
190
191     [MethodImpl(MethodImplOptions.AggressiveInlining)]
192     public void CopyTo(TLink[] array, int arrayIndex)
193     {
194         Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
195         Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
196             ↪ nameof(arrayIndex));
197         if (arrayIndex + Length > array.Length)
198         {
199             throw new InvalidOperationException();
200         }
201         array[arrayIndex++] = Index;
202         array[arrayIndex++] = Source;
203         array[arrayIndex] = Target;
204     }
205
206     [MethodImpl(MethodImplOptions.AggressiveInlining)]
207     public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
208
209     [MethodImpl(MethodImplOptions.AggressiveInlining)]
210     public int IndexOf(TLink item)
211     {
212         if (_equalityComparer.Equals(Index, item))
213         {
214             return _constants.IndexPart;
215         }
216     }

```

```

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

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets
6  {
7      public static class LinkExtensions
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
11             ↪ Point<TLink>.IsFullPoint(link);
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
15             ↪ Point<TLink>.IsPartialPoint(link);
16     }
17 }

```

### 1.24 ./csharp/Platform.Data.Doublets/LinksOperatorBase.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets
6  {
7      public abstract class LinksOperatorBase<TLink>
8      {
9          protected readonly ILinks<TLink> _links;
10
11          public ILinks<TLink> Links
12          {
13              [MethodImpl(MethodImplOptions.AggressiveInlining)]
14              get => _links;
15          }
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected LinksOperatorBase(ILinks<TLink> links) => _links = links;
19      }
20 }

```

### 1.25 ./csharp/Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs

```

1  using System.Collections.Generic;
2  using Platform.Reflection;
3  using Platform.Converters;
4  using Platform.Numbers;
5  using System.Runtime.CompilerServices;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary

```

```

10 {
11     public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
        ↳ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
        ↳ EqualityComparer<TLink>.Default;
14         private static readonly TLink _zero = default;
15         private static readonly TLink _one = Arithmetic.Increment(_zero);
16
17         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
        ↳ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
        ↳ powerOf2ToUnaryNumberConverter;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public TLink Convert(TLink number)
24         {
25             var links = _links;
26             var nullConstant = links.Constants.Null;
27             var target = nullConstant;
28             for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
        ↳ NumericType<TLink>.BitsSize; i++)
29             {
30                 if (_equalityComparer.Equals(Bit.And(number, _one), _one))
31                 {
32                     target = _equalityComparer.Equals(target, nullConstant)
33                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
34                         : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
35                 }
36                 number = Bit.ShiftRight(number, 1);
37             }
38             return target;
39         }
40     }
41 }

```

## 1.26 ./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConverter.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 LinkToItsFrequencyNumberConverter<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 LinkToItsFrequencyNumberConverter(
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 }

```

### 1.27 ./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>,
12         ↪ IConverter<int, TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↪ EqualityComparer<TLink>.Default;
16
17         private readonly TLink[] _unaryNumberPowersOf2;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
21         {
22             _unaryNumberPowersOf2 = new TLink[64];
23             _unaryNumberPowersOf2[0] = one;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(int power)
28         {
29             Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
30                 ↪ - 1), nameof(power));
31             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
32             {
33                 return _unaryNumberPowersOf2[power];
34             }
35             var previousPowerOf2 = Convert(power - 1);
36             var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
37             _unaryNumberPowersOf2[power] = powerOf2;
38             return powerOf2;
39         }
40     }
41 }

```

### 1.28 ./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>,
11         ↪ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
16             ↪ UncheckedConverter<TLink, ulong>.Default;
17         private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
18             ↪ UncheckedConverter<ulong, TLink>.Default;
19         private static readonly TLink _zero = default;
20         private static readonly TLink _one = Arithmetic.Increment(_zero);
21
22         private readonly Dictionary<TLink, TLink> _unaryToUInt64;
23         private readonly TLink _unaryOne;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
27             : base(links)
28         {
29         }
30     }
31 }

```

```

24 {
25     _unaryOne = unaryOne;
26     _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
27 }
28
29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 public TLink Convert(TLink unaryNumber)
31 {
32     if (_equalityComparer.Equals(unaryNumber, default))
33     {
34         return default;
35     }
36     if (_equalityComparer.Equals(unaryNumber, _unaryOne))
37     {
38         return _one;
39     }
40     var links = _links;
41     var source = links.GetSource(unaryNumber);
42     var target = links.GetTarget(unaryNumber);
43     if (_equalityComparer.Equals(source, target))
44     {
45         return _unaryToUInt64[unaryNumber];
46     }
47     else
48     {
49         var result = _unaryToUInt64[source];
50         TLink lastValue;
51         while (!_unaryToUInt64.TryGetValue(target, out lastValue))
52         {
53             source = links.GetSource(target);
54             result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
55             target = links.GetTarget(target);
56         }
57         result = Arithmetic<TLink>.Add(result, lastValue);
58         return result;
59     }
60 }
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
↪ links, TLink unaryOne)
64 {
65     var unaryToUInt64 = new Dictionary<TLink, TLink>
66     {
67         { unaryOne, _one }
68     };
69     var unary = unaryOne;
70     var number = _one;
71     for (var i = 1; i < 64; i++)
72     {
73         unary = links.GetOrCreate(unary, unary);
74         number = Double(number);
75         unaryToUInt64.Add(unary, number);
76     }
77     return unaryToUInt64;
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 private static TLink Double(TLink number) =>
↪ _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
82 }
83 }

```

## 1.29 ./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Reflection;
4 using Platform.Converters;
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.Numbers.Unary
10 {
11     public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
↪ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
↪ EqualityComparer<TLink>.Default;
14         private static readonly TLink _zero = default;

```



```

15     private static readonly TLink _one = Arithmetic.Increment(_zero);
16
17     private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
        ↳ TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
        ↳ = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     public TLink Convert(TLink sourceNumber)
24     {
25         var links = _links;
26         var nullConstant = links.Constants.Null;
27         var source = sourceNumber;
28         var target = nullConstant;
29         if (!_equalityComparer.Equals(source, nullConstant))
30         {
31             while (true)
32             {
33                 if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
34                 {
35                     SetBit(ref target, powerOf2Index);
36                     break;
37                 }
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 }

```

### 1.30 ./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs

```

1  using System.Linq;
2  using System.Collections.Generic;
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
8  namespace Platform.Data.Doublets.PropertyOperators
9  {
10     public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>, IProperties<TLink, TLink,
        ↳ TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public TLink GetValue(TLink @object, TLink property)
19         {
20             var links = _links;
21             var objectProperty = links.SearchOrDefault(@object, property);
22             if (_equalityComparer.Equals(objectProperty, default))

```

```

23     {
24         return default;
25     }
26     var constants = links.Constants;
27     var valueLink = links.All(constants.Any, objectProperty).SingleOrDefault();
28     if (valueLink == null)
29     {
30         return default;
31     }
32     return links.GetTarget(valueLink[constants.IndexPart]);
33 }
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 public void SetValue(TLink @object, TLink property, TLink value)
37 {
38     var links = _links;
39     var objectProperty = links.GetOrCreate(@object, property);
40     links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
41     links.GetOrCreate(objectProperty, value);
42 }
43 }
44 }

```

### 1.31 ./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.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 PropertyOperator<TLink> : LinksOperatorBase<TLink>, IProperty<TLink, TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↳ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _propertyMarker;
15         private readonly TLink _propertyValueMarker;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
19             ↳ propertyValueMarker) : base(links)
20         {
21             _propertyMarker = propertyMarker;
22             _propertyValueMarker = propertyValueMarker;
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public TLink Get(TLink link)
27         {
28             var property = _links.SearchOrDefault(link, _propertyMarker);
29             return GetValue(GetContainer(property));
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         private TLink GetContainer(TLink property)
34         {
35             var valueContainer = default(TLink);
36             if (_equalityComparer.Equals(property, default))
37             {
38                 return valueContainer;
39             }
40             var links = _links;
41             var constants = links.Constants;
42             var countinueConstant = constants.Continue;
43             var breakConstant = constants.Break;
44             var anyConstant = constants.Any;
45             var query = new Link<TLink>(anyConstant, property, anyConstant);
46             links.Each(candidate =>
47             {
48                 var candidateTarget = links.GetTarget(candidate);
49                 var valueTarget = links.GetTarget(candidateTarget);
50                 if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
51                 {
52                     valueContainer = links.GetIndex(candidate);
53                     return breakConstant;
54                 }
55                 return countinueConstant;
56             }, query);
57         }
58     }
59 }

```

```

55     return valueContainer;
56 }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
    ↪ ? default : _links.GetTarget(container);
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 public void Set(TLink link, TLink value)
63 {
64     var links = _links;
65     var property = links.GetOrCreate(link, _propertyMarker);
66     var container = GetContainer(property);
67     if (_equalityComparer.Equals(container, default))
68     {
69         links.GetOrCreate(property, value);
70     }
71     else
72     {
73         links.Update(container, property, value);
74     }
75 }
76 }
77 }

```

### 1.32 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksAvlBalancedTreeMethodsBase.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using Platform.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.ResizableDirectMemory.Generic
13 {
14     public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> :
15     ↪ SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
16     {
17         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
18         ↪ UncheckedConverter<TLink, long>.Default;
19         private static readonly UncheckedConverter<TLink, int> _addressToInt32Converter =
20         ↪ UncheckedConverter<TLink, int>.Default;
21         private static readonly UncheckedConverter<bool, TLink> _boolToAddressConverter =
22         ↪ UncheckedConverter<bool, TLink>.Default;
23         private static readonly UncheckedConverter<TLink, bool> _addressToBoolConverter =
24         ↪ UncheckedConverter<TLink, bool>.Default;
25         private static readonly UncheckedConverter<int, TLink> _int32ToAddressConverter =
26         ↪ UncheckedConverter<int, TLink>.Default;
27
28         protected readonly TLink Break;
29         protected readonly TLink Continue;
30         protected readonly byte* Links;
31         protected readonly byte* Header;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
35         ↪ byte* header)
36         {
37             Links = links;
38             Header = header;
39             Break = constants.Break;
40             Continue = constants.Continue;
41         }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected abstract TLink GetTreeRoot();
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected abstract TLink GetBasePartValue(TLink link);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
51         ↪ rootSource, TLink rootTarget);
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
55         ↪ rootSource, TLink rootTarget);
56     }
57 }

```

```

47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
49     ↳ 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 FirstIsToLeftOfSecond(TLink first, TLink second)
63 {
64     ref var firstLink = ref GetLinkReference(first);
65     ref var secondLink = ref GetLinkReference(second);
66     return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
    ↳ secondLink.Source, secondLink.Target);
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
71 {
72     ref var firstLink = ref GetLinkReference(first);
73     ref var secondLink = ref GetLinkReference(second);
74     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↳ secondLink.Source, secondLink.Target);
75 }
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
    ↳ -5);
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
    ↳ Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected virtual bool GetLeftIsChildValue(TLink value)
85 {
86     unchecked
87     {
88         return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 4, 1));
89         //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
90     }
91 }
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
95 {
96     unchecked
97     {
98         var previousValue = storedValue;
99         var modified = Bit<TLink>.PartialWrite(previousValue,
    ↳ _boolToAddressConverter.Convert(value), 4, 1);
100         storedValue = modified;
101     }
102 }
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 protected virtual bool GetRightIsChildValue(TLink value)
106 {
107     unchecked
108     {
109         return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 3, 1));
110         //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
111     }
112 }
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
116 {

```

```

117         unchecked
118         {
119             var previousValue = storedValue;
120             var modified = Bit<TLink>.PartialWrite(previousValue,
121                 ↪ _boolToAddressConverter.Convert(value), 3, 1);
122             storedValue = modified;
123         }
124     }
125     [MethodImpl(MethodImplOptions.AggressiveInlining)]
126     protected bool IsChild(TLink parent, TLink possibleChild)
127     {
128         var parentSize = GetSize(parent);
129         var childSize = GetSizeOrZero(possibleChild);
130         return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
131     }
132
133     [MethodImpl(MethodImplOptions.AggressiveInlining)]
134     protected virtual sbyte GetBalanceValue(TLink storedValue)
135     {
136         unchecked
137         {
138             var value = _addressToInt32Converter.Convert(Bit<TLink>.PartialRead(storedValue,
139                 ↪ 0, 3));
140             value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
141                 ↪ end of sbyte
142             return (sbyte)value;
143         }
144     }
145     [MethodImpl(MethodImplOptions.AggressiveInlining)]
146     protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
147     {
148         unchecked
149         {
150             var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
151                 ↪ value & 3);
152             var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
153             storedValue = modified;
154         }
155     }
156
157     public TLink this[TLink index]
158     {
159         [MethodImpl(MethodImplOptions.AggressiveInlining)]
160         get
161         {
162             var root = GetTreeRoot();
163             if (GreaterOrEqualThan(index, GetSize(root)))
164             {
165                 return Zero;
166             }
167             while (!EqualToZero(root))
168             {
169                 var left = GetLeftOrDefault(root);
170                 var leftSize = GetSizeOrZero(left);
171                 if (LessThan(index, leftSize))
172                 {
173                     root = left;
174                     continue;
175                 }
176                 if (AreEqual(index, leftSize))
177                 {
178                     return root;
179                 }
180                 root = GetRightOrDefault(root);
181                 index = Subtract(index, Increment(leftSize));
182             }
183             return Zero; // TODO: Impossible situation exception (only if tree structure
184                 ↪ broken)
185         }
186     }
187
188     /// <summary>
189     /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
190     /// ↪ (концом).
191     /// </summary>
192     /// <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)) //
201             ↪ node.Key < root.Key
202         {
203             root = GetLeftOrDefault(root);
204         }
205         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
206             ↪ node.Key > root.Key
207         {
208             root = GetRightOrDefault(root);
209         }
210         else // node.Key == root.Key
211         {
212             return root;
213         }
214     }
215     return Zero;
216 }
217
218 // TODO: Return indices range instead of references count
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 public TLink CountUsages(TLink link)
221 {
222     var root = GetTreeRoot();
223     var total = GetSize(root);
224     var totalRightIgnore = Zero;
225     while (!EqualToZero(root))
226     {
227         var @base = GetBasePartValue(root);
228         if (LessOrEqualThan(@base, link))
229         {
230             root = GetRightOrDefault(root);
231         }
232         else
233         {
234             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
235             root = GetLeftOrDefault(root);
236         }
237     }
238     root = GetTreeRoot();
239     var totalLeftIgnore = Zero;
240     while (!EqualToZero(root))
241     {
242         var @base = GetBasePartValue(root);
243         if (GreaterOrEqualThan(@base, link))
244         {
245             root = GetLeftOrDefault(root);
246         }
247         else
248         {
249             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
250             root = GetRightOrDefault(root);
251         }
252     }
253     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
254 }
255
256 [MethodImpl(MethodImplOptions.AggressiveInlining)]
257 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
258 {
259     var root = GetTreeRoot();
260     if (EqualToZero(root))
261     {
262         return Continue;
263     }
264     TLink first = Zero, current = root;
265     while (!EqualToZero(current))
266     {

```

```

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 }

```

### 1.33 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSizeBalancedTreeMethodsBase.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using 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.ResizableDirectMemory.Generic
12 {
13     public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> :
14         ↳ SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
17             ↳ UncheckedConverter<TLink, long>.Default;
18
19         protected readonly TLink Break;
20         protected readonly TLink Continue;
21         protected readonly byte* Links;
22         protected readonly byte* Header;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
26             ↳ byte* header)
27         {
28             Links = links;
29             Header = header;
30             Break = constants.Break;
31             Continue = constants.Continue;
32         }
33     }
34 }

```

```

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)]
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
104   ↳ (концом).
105 /// </summary>
106 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
107 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
108 /// <returns>Индекс искомой связи.</returns>
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 public TLink Search(TLink source, TLink target)
111 {
112     var root = GetTreeRoot();
113     while (!EqualToZero(root))
114     {
115         ref var rootLink = ref GetLinkReference(root);
116         var rootSource = rootLink.Source;
117         var rootTarget = rootLink.Target;
118         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
119             ↳ node.Key < root.Key
120         {
121             root = GetLeftOrDefault(root);
122         }
123         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
124             ↳ node.Key > root.Key
125         {
126             root = GetRightOrDefault(root);
127         }
128         else // node.Key == root.Key
129         {
130             return root;
131         }
132     }
133     return Zero;
134 }
135
136 // TODO: Return indices range instead of references count
137 [MethodImpl(MethodImplOptions.AggressiveInlining)]
138 public TLink CountUsages(TLink link)
139 {
140     var root = GetTreeRoot();
141     var total = GetSize(root);
142     var totalRightIgnore = Zero;
143     while (!EqualToZero(root))
144     {
145         var @base = GetBasePartValue(root);
146         if (LessOrEqualThan(@base, link))
147         {
148             root = GetRightOrDefault(root);
149         }
150         else
151         {
152             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
153             root = GetLeftOrDefault(root);
154         }
155     }
156     root = GetTreeRoot();
157     var totalLeftIgnore = Zero;
158     while (!EqualToZero(root))
159     {
160         var @base = GetBasePartValue(root);
161         if (GreaterOrEqualThan(@base, link))
162         {
163             root = GetLeftOrDefault(root);
164         }
165         else
166         {
167             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
168             root = GetRightOrDefault(root);
169         }
170     }
171     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
172 }
173
174 [MethodImpl(MethodImplOptions.AggressiveInlining)]
175 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
176     ↳ EachUsageCore(@base, GetTreeRoot(), handler);
177
178 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
179     ↳ 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     {
194         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
195         {
196             return @break;
197         }
198     }
199     else //if (linkBasePart == @base)
200     {
201         if (AreEqual(handler(GetLinkValues(link)), @break))
202         {
203             return @break;
204         }
205         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
206         {
207             return @break;
208         }
209         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
210         {
211             return @break;
212         }
213     }
214     return @continue;
215 }
216
217 [MethodImpl(MethodImplOptions.AggressiveInlining)]
218 protected override void PrintNodeValue(TLink node, StringBuilder sb)
219 {
220     ref var link = ref GetLinkReference(node);
221     sb.Append(' ');
222     sb.Append(link.Source);
223     sb.Append('-');
224     sb.Append('>');
225     sb.Append(link.Target);
226 }
227 }
228 }

```

### 1.34 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesAvlBalancedTreeMethods.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.ResizableDirectMemory.Generic
6 {
7     public unsafe class LinksSourcesAvlBalancedTreeMethods<TLink> :
8         ↳ LinksAvlBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksSourcesAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
12             ↳ byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref TLink GetLeftReference(TLink node) => ref
16             ↳ GetLinkReference(node).LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetRightReference(TLink node) => ref
20             ↳ GetLinkReference(node).RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void PrintNodeValue(TLink node, StringBuilder sb)
30         {
31             ref var link = ref GetLinkReference(node);
32             sb.Append(' ');
33             sb.Append(link.Source);
34             sb.Append('-');
35             sb.Append('>');
36             sb.Append(link.Target);
37         }
38     }
39 }

```

```

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) =>
26     ↳ GetLinkReference(node).LeftAsSource = left;
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 protected override void SetRight(TLink node, TLink right) =>
30     ↳ GetLinkReference(node).RightAsSource = right;
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 protected override TLink GetSize(TLink node) =>
34     ↳ GetSizeValue(GetLinkReference(node).SizeAsSource);
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
38     ↳ GetLinkReference(node).SizeAsSource, size);
39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 protected override bool GetLeftIsChild(TLink node) =>
42     ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 protected override void SetLeftIsChild(TLink node, bool value) =>
46     ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected override bool GetRightIsChild(TLink node) =>
50     ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsSource);
51
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 protected override void SetRightIsChild(TLink node, bool value) =>
54     ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override sbyte GetBalance(TLink node) =>
58     ↳ GetBalanceValue(GetLinkReference(node).SizeAsSource);
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
62     ↳ GetLinkReference(node).SizeAsSource, value);
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsSource;
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
72     ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
73     ↳ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
77     ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
78     ↳ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 protected override void ClearNode(TLink node)
82 {
83     ref var link = ref GetLinkReference(node);
84     link.LeftAsSource = Zero;
85     link.RightAsSource = Zero;
86     link.SizeAsSource = Zero;
87 }
88 }
89 }

```

### 1.35 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesSizeBalancedTreeMethods.cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
6 {

```

```

7 public unsafe class LinksSourcesSizeBalancedTreeMethods<TLink> :
  ↳ LinksSizeBalancedTreeMethodsBase<TLink>
8 {
9     [MethodImpl(MethodImplOptions.AggressiveInlining)]
10    public LinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
  ↳ byte* header) : base(constants, links, header) { }
11
12    [MethodImpl(MethodImplOptions.AggressiveInlining)]
13    protected override ref TLink GetLeftReference(TLink node) => ref
  ↳ GetLinkReference(node).LeftAsSource;
14
15    [MethodImpl(MethodImplOptions.AggressiveInlining)]
16    protected override ref TLink GetRightReference(TLink node) => ref
  ↳ GetLinkReference(node).RightAsSource;
17
18    [MethodImpl(MethodImplOptions.AggressiveInlining)]
19    protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
20
21    [MethodImpl(MethodImplOptions.AggressiveInlining)]
22    protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
23
24    [MethodImpl(MethodImplOptions.AggressiveInlining)]
25    protected override void SetLeft(TLink node, TLink left) =>
  ↳ GetLinkReference(node).LeftAsSource = left;
26
27    [MethodImpl(MethodImplOptions.AggressiveInlining)]
28    protected override void SetRight(TLink node, TLink right) =>
  ↳ GetLinkReference(node).RightAsSource = right;
29
30    [MethodImpl(MethodImplOptions.AggressiveInlining)]
31    protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsSource;
32
33    [MethodImpl(MethodImplOptions.AggressiveInlining)]
34    protected override void SetSize(TLink node, TLink size) =>
  ↳ GetLinkReference(node).SizeAsSource = size;
35
36    [MethodImpl(MethodImplOptions.AggressiveInlining)]
37    protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsSource;
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 }

```

### 1.36 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsAvlBalancedTreeMethods.cs

```

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

```

```

13     protected override ref TLink GetLeftReference(TLink node) => ref
14         ↳ GetLinkReference(node).LeftAsTarget;
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     protected override ref TLink GetRightReference(TLink node) => ref
18         ↳ GetLinkReference(node).RightAsTarget;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override void SetLeft(TLink node, TLink left) =>
28         ↳ GetLinkReference(node).LeftAsTarget = left;
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected override void SetRight(TLink node, TLink right) =>
32         ↳ GetLinkReference(node).RightAsTarget = right;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override TLink GetSize(TLink node) =>
36         ↳ GetSizeValue(GetLinkReference(node).SizeAsTarget);
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
40         ↳ GetLinkReference(node).SizeAsTarget, size);
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override bool GetLeftIsChild(TLink node) =>
44         ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override void SetLeftIsChild(TLink node, bool value) =>
48         ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override bool GetRightIsChild(TLink node) =>
52         ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override void SetRightIsChild(TLink node, bool value) =>
56         ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     protected override sbyte GetBalance(TLink node) =>
60         ↳ GetBalanceValue(GetLinkReference(node).SizeAsTarget);
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
64         ↳ GetLinkReference(node).SizeAsTarget, value);
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
74         ↳ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
75         ↳ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
76
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
79         ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
80         ↳ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected override void ClearNode(TLink node)
84     {
85         ref var link = ref GetLinkReference(node);
86         link.LeftAsTarget = Zero;
87         link.RightAsTarget = Zero;
88         link.SizeAsTarget = Zero;
89     }
86 }

```

75 }

### 1.37 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsSizeBalancedTreeMethods.cs

```
1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
6 {
7     public unsafe class LinksTargetsSizeBalancedTreeMethods<TLink> :
8         ↳ LinksSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
12             ↳ byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref TLink GetLeftReference(TLink node) => ref
16             ↳ GetLinkReference(node).LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetRightReference(TLink node) => ref
20             ↳ GetLinkReference(node).RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(TLink node, TLink left) =>
30             ↳ GetLinkReference(node).LeftAsTarget = left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(TLink node, TLink right) =>
34             ↳ GetLinkReference(node).RightAsTarget = right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(TLink node, TLink size) =>
41             ↳ GetLinkReference(node).SizeAsTarget = size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
51             ↳ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
52             ↳ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
56             ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
57             ↳ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override void ClearNode(TLink node)
61         {
62             ref var link = ref GetLinkReference(node);
63             link.LeftAsTarget = Zero;
64             link.RightAsTarget = Zero;
65             link.SizeAsTarget = Zero;
66         }
67     }
68 }
```

### 1.38 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinks.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.ResizableDirectMemory.Generic
10 {
11     public unsafe class ResizableDirectMemoryLinks<TLink> : ResizableDirectMemoryLinksBase<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 ResizableDirectMemoryLinks(string address) : this(address, DefaultLinksSizeStep)
20         { }
21
22         /// <summary>
23         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
24         /// минимальным шагом расширения базы данных.
25         /// </summary>
26         /// <param name="address">Полный путь к файлу базы данных.</param>
27         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
28         /// байтах.</param>
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public ResizableDirectMemoryLinks(string address, long memoryReservationStep) : this(new
31         { }
32         { }
33         { }
34         { }
35         { }
36         { }
37         { }
38         { }
39         { }
40         { }
41         { }
42         { }
43         { }
44         { }
45         { }
46         { }
47         { }
48         { }
49         { }
50         { }
51         { }
52         { }
53         { }
54         { }
55         { }
56         { }
57         { }
58         { }
59         { }
60         { }
61         { }
62         { }
63         { }
64         { }
65         { }
66         { }
67         { }
68         { }
69         { }
70         { }
71         { }
72         { }
73         { }
74         { }
75         { }
76         { }
77         { }
78         { }
79         { }
80         { }
81         { }
82         { }
83         { }
84         { }
85         { }
86         { }
87         { }
88         { }
89         { }
90         { }
91         { }
92         { }
93         { }
94         { }
95         { }
96         { }
97         { }
98         { }
99         { }
100        { }
101        { }
102        { }
103        { }
104        { }
105        { }
106        { }
107        { }
108        { }
109        { }
110        { }
111        { }
112        { }
113        { }
114        { }
115        { }
116        { }
117        { }
118        { }
119        { }
120        { }
121        { }
122        { }
123        { }
124        { }
125        { }
126        { }
127        { }
128        { }
129        { }
130        { }
131        { }
132        { }
133        { }
134        { }
135        { }
136        { }
137        { }
138        { }
139        { }
140        { }
141        { }
142        { }
143        { }
144        { }
145        { }
146        { }
147        { }
148        { }
149        { }
150        { }
151        { }
152        { }
153        { }
154        { }
155        { }
156        { }
157        { }
158        { }
159        { }
160        { }
161        { }
162        { }
163        { }
164        { }
165        { }
166        { }
167        { }
168        { }
169        { }
170        { }
171        { }
172        { }
173        { }
174        { }
175        { }
176        { }
177        { }
178        { }
179        { }
180        { }
181        { }
182        { }
183        { }
184        { }
185        { }
186        { }
187        { }
188        { }
189        { }
190        { }
191        { }
192        { }
193        { }
194        { }
195        { }
196        { }
197        { }
198        { }
199        { }
200        { }
201        { }
202        { }
203        { }
204        { }
205        { }
206        { }
207        { }
208        { }
209        { }
210        { }
211        { }
212        { }
213        { }
214        { }
215        { }
216        { }
217        { }
218        { }
219        { }
220        { }
221        { }
222        { }
223        { }
224        { }
225        { }
226        { }
227        { }
228        { }
229        { }
230        { }
231        { }
232        { }
233        { }
234        { }
235        { }
236        { }
237        { }
238        { }
239        { }
240        { }
241        { }
242        { }
243        { }
244        { }
245        { }
246        { }
247        { }
248        { }
249        { }
250        { }
251        { }
252        { }
253        { }
254        { }
255        { }
256        { }
257        { }
258        { }
259        { }
260        { }
261        { }
262        { }
263        { }
264        { }
265        { }
266        { }
267        { }
268        { }
269        { }
270        { }
271        { }
272        { }
273        { }
274        { }
275        { }
276        { }
277        { }
278        { }
279        { }
280        { }
281        { }
282        { }
283        { }
284        { }
285        { }
286        { }
287        { }
288        { }
289        { }
290        { }
291        { }
292        { }
293        { }
294        { }
295        { }
296        { }
297        { }
298        { }
299        { }
300        { }
301        { }
302        { }
303        { }
304        { }
305        { }
306        { }
307        { }
308        { }
309        { }
310        { }
311        { }
312        { }
313        { }
314        { }
315        { }
316        { }
317        { }
318        { }
319        { }
320        { }
321        { }
322        { }
323        { }
324        { }
325        { }
326        { }
327        { }
328        { }
329        { }
330        { }
331        { }
332        { }
333        { }
334        { }
335        { }
336        { }
337        { }
338        { }
339        { }
340        { }
341        { }
342        { }
343        { }
344        { }
345        { }
346        { }
347        { }
348        { }
349        { }
350        { }
351        { }
352        { }
353        { }
354        { }
355        { }
356        { }
357        { }
358        { }
359        { }
360        { }
361        { }
362        { }
363        { }
364        { }
365        { }
366        { }
367        { }
368        { }
369        { }
370        { }
371        { }
372        { }
373        { }
374        { }
375        { }
376        { }
377        { }
378        { }
379        { }
380        { }
381        { }
382        { }
383        { }
384        { }
385        { }
386        { }
387        { }
388        { }
389        { }
390        { }
391        { }
392        { }
393        { }
394        { }
395        { }
396        { }
397        { }
398        { }
399        { }
400        { }
401        { }
402        { }
403        { }
404        { }
405        { }
406        { }
407        { }
408        { }
409        { }
410        { }
411        { }
412        { }
413        { }
414        { }
415        { }
416        { }
417        { }
418        { }
419        { }
420        { }
421        { }
422        { }
423        { }
424        { }
425        { }
426        { }
427        { }
428        { }
429        { }
430        { }
431        { }
432        { }
433        { }
434        { }
435        { }
436        { }
437        { }
438        { }
439        { }
440        { }
441        { }
442        { }
443        { }
444        { }
445        { }
446        { }
447        { }
448        { }
449        { }
450        { }
451        { }
452        { }
453        { }
454        { }
455        { }
456        { }
457        { }
458        { }
459        { }
460        { }
461        { }
462        { }
463        { }
464        { }
465        { }
466        { }
467        { }
468        { }
469        { }
470        { }
471        { }
472        { }
473        { }
474        { }
475        { }
476        { }
477        { }
478        { }
479        { }
480        { }
481        { }
482        { }
483        { }
484        { }
485        { }
486        { }
487        { }
488        { }
489        { }
490        { }
491        { }
492        { }
493        { }
494        { }
495        { }
496        { }
497        { }
498        { }
499        { }
500        { }
501        { }
502        { }
503        { }
504        { }
505        { }
506        { }
507        { }
508        { }
509        { }
510        { }
511        { }
512        { }
513        { }
514        { }
515        { }
516        { }
517        { }
518        { }
519        { }
520        { }
521        { }
522        { }
523        { }
524        { }
525        { }
526        { }
527        { }
528        { }
529        { }
530        { }
531        { }
532        { }
533        { }
534        { }
535        { }
536        { }
537        { }
538        { }
539        { }
540        { }
541        { }
542        { }
543        { }
544        { }
545        { }
546        { }
547        { }
548        { }
549        { }
550        { }
551        { }
552        { }
553        { }
554        { }
555        { }
556        { }
557        { }
558        { }
559        { }
560        { }
561        { }
562        { }
563        { }
564        { }
565        { }
566        { }
567        { }
568        { }
569        { }
570        { }
571        { }
572        { }
573        { }
574        { }
575        { }
576        { }
577        { }
578        { }
579        { }
580        { }
581        { }
582        { }
583        { }
584        { }
585        { }
586        { }
587        { }
588        { }
589        { }
590        { }
591        { }
592        { }
593        { }
594        { }
595        { }
596        { }
597        { }
598        { }
599        { }
600        { }
601        { }
602        { }
603        { }
604        { }
605        { }
606        { }
607        { }
608        { }
609        { }
610        { }
611        { }
612        { }
613        { }
614        { }
615        { }
616        { }
617        { }
618        { }
619        { }
620        { }
621        { }
622        { }
623        { }
624        { }
625        { }
626        { }
627        { }
628        { }
629        { }
630        { }
631        { }
632        { }
633        { }
634        { }
635        { }
636        { }
637        { }
638        { }
639        { }
640        { }
641        { }
642        { }
643        { }
644        { }
645        { }
646        { }
647        { }
648        { }
649        { }
650        { }
651        { }
652        { }
653        { }
654        { }
655        { }
656        { }
657        { }
658        { }
659        { }
660        { }
661        { }
662        { }
663        { }
664        { }
665        { }
666        { }
667        { }
668        { }
669        { }
670        { }
671        { }
672        { }
673        { }
674        { }
675        { }
676        { }
677        { }
678        { }
679        { }
680        { }
681        { }
682        { }
683        { }
684        { }
685        { }
686        { }
687        { }
688        { }
689        { }
690        { }
691        { }
692        { }
693        { }
694        { }
695        { }
696        { }
697        { }
698        { }
699        { }
700        { }
701        { }
702        { }
703        { }
704        { }
705        { }
706        { }
707        { }
708        { }
709        { }
710        { }
711        { }
712        { }
713        { }
714        { }
715        { }
716        { }
717        { }
718        { }
719        { }
720        { }
721        { }
722        { }
723        { }
724        { }
725        { }
726        { }
727        { }
728        { }
729        { }
730        { }
731        { }
732        { }
733        { }
734        { }
735        { }
736        { }
737        { }
738        { }
739        { }
740        { }
741        { }
742        { }
743        { }
744        { }
745        { }
746        { }
747        { }
748        { }
749        { }
750        { }
751        { }
752        { }
753        { }
754        { }
755        { }
756        { }
757        { }
758        { }
759        { }
760        { }
761        { }
762        { }
763        { }
764        { }
765        { }
766        { }
767        { }
768        { }
769        { }
770        { }
771        { }
772        { }
773        { }
774        { }
775        { }
776        { }
777        { }
778        { }
779        { }
780        { }
781        { }
782        { }
783        { }
784        { }
785        { }
786        { }
787        { }
788        { }
789        { }
790        { }
791        { }
792        { }
793        { }
794        { }
795        { }
796        { }
797        { }
798        { }
799        { }
800        { }
801        { }
802        { }
803        { }
804        { }
805        { }
806        { }
807        { }
808        { }
809        { }
810        { }
811        { }
812        { }
813        { }
814        { }
815        { }
816        { }
817        { }
818        { }
819        { }
820        { }
821        { }
822        { }
823        { }
824        { }
825        { }
826        { }
827        { }
828        { }
829        { }
830        { }
831        { }
832        { }
833        { }
834        { }
835        { }
836        { }
837        { }
838        { }
839        { }
840        { }
841        { }
842        { }
843        { }
844        { }
845        { }
846        { }
847        { }
848        { }
849        { }
850        { }
851        { }
852        { }
853        { }
854        { }
855        { }
856        { }
857        { }
858        { }
859        { }
860        { }
861        { }
862        { }
863        { }
864        { }
865        { }
866        { }
867        { }
868        { }
869        { }
870        { }
871        { }
872        { }
873        { }
874        { }
875        { }
876        { }
877        { }
878        { }
879        { }
880        { }
881        { }
882        { }
883        { }
884        { }
885        { }
886        { }
887        { }
888        { }
889        { }
890        { }
891        { }
892        { }
893        { }
894        { }
895        { }
896        { }
897        { }
898        { }
899        { }
900        { }
901        { }
902        { }
903        { }
904        { }
905        { }
906        { }
907        { }
908        { }
909        { }
910        { }
911        { }
912        { }
913        { }
914        { }
915        { }
916        { }
917        { }
918        { }
919        { }
920        { }
921        { }
922        { }
923        { }
924        { }
925        { }
926        { }
927        { }
928        { }
929        { }
930        { }
931        { }
932        { }
933        { }
934        { }
935        { }
936        { }
937        { }
938        { }
939        { }
940        { }
941        { }
942        { }
943        { }
944        { }
945        { }
946        { }
947        { }
948        { }
949        { }
950        { }
951        { }
952        { }
953        { }
954        { }
955        { }
956        { }
957        { }
958        { }
959        { }
960        { }
961        { }
962        { }
963        { }
964        { }
965        { }
966        { }
967        { }
968        { }
969        { }
970        { }
971        { }
972        { }
973        { }
974        { }
975        { }
976        { }
977        { }
978        { }
979        { }
980        { }
981        { }
982        { }
983        { }
984        { }
985        { }
986        { }
987        { }
988        { }
989        { }
990        { }
991        { }
992        { }
993        { }
994        { }
995        { }
996        { }
997        { }
998        { }
999        { }
1000       { }

```

```

70     protected override ref LinkHeader<TLink> GetHeaderReference() => ref
71         ↳ AsRef<LinkHeader<TLink>>(_header);
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
75         ↳ AsRef<RawLink<TLink>>(_links + (LinkSizeInBytes * ConvertToInt64(linkIndex)));
76 }

```

### 1.39 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinksBase.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Singletons;
6  using Platform.Converters;
7  using Platform.Numbers;
8  using Platform.Memory;
9  using Platform.Data.Exceptions;
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
14 {
15     public abstract class ResizableDirectMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
18             ↳ EqualityComparer<TLink>.Default;
19         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
20         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
21             ↳ UncheckedConverter<TLink, long>.Default;
22         private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
23             ↳ UncheckedConverter<long, TLink>.Default;
24
25         private static readonly TLink _zero = default;
26         private static readonly TLink _one = Arithmetic.Increment(_zero);
27
28         /// <summary>Возвращает размер одной связи в байтах.</summary>
29         /// <remarks>
30         ///     Используется только во вне класса, не рекомендуется использовать внутри.
31         ///     Так как во вне не обязательно будет доступен unsafe C#.
32         /// </remarks>
33         public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
34
35         public static readonly long LinkHeaderSizeInBytes = LinkHeader<TLink>.SizeInBytes;
36
37         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
38
39         protected readonly IResizableDirectMemory _memory;
40         protected readonly long _memoryReservationStep;
41
42         protected ILinksTreeMethods<TLink> TargetsTreeMethods;
43         protected ILinksTreeMethods<TLink> SourcesTreeMethods;
44         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
45         // ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
46         // ↳ наличие связи внутри
47         protected ILinksListMethods<TLink> UnusedLinksListMethods;
48
49         /// <summary>
50         ///     Возвращает общее число связей находящихся в хранилище.
51         /// </summary>
52         protected virtual TLink Total
53         {
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             get
56             {
57                 ref var header = ref GetHeaderReference();
58                 return Subtract(header.AllocatedLinks, header.FreeLinks);
59             }
60         }
61
62         public virtual LinksConstants<TLink> Constants
63         {
64             [MethodImpl(MethodImplOptions.AggressiveInlining)]
65             get;
66         }
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         protected ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
70             ↳ memoryReservationStep, LinksConstants<TLink> constants)
71         {

```



```

66     _memory = memory;
67     _memoryReservationStep = memoryReservationStep;
68     Constants = constants;
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 protected ResizableDirectMemoryLinksBase(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 = (Convert.ToInt64(header.AllocatedLinks) * LinkSizeInBytes) +
    ↪ LinkHeaderSizeInBytes;
85     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
86     header.ReservedLinks = Convert.ToInt64((memory.ReservedCapacity -
    ↪ LinkHeaderSizeInBytes) / LinkSizeInBytes);
87 }
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 public virtual TLink Count(IList<TLink> restrictions)
91 {
92     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
93     if (restrictions.Count == 0)
94     {
95         return Total;
96     }
97     var constants = Constants;
98     var any = constants.Any;
99     var index = restrictions[constants.IndexPart];
100     if (restrictions.Count == 1)
101     {
102         if (AreEqual(index, any))
103         {
104             return Total;
105         }
106         return Exists(index) ? GetOne() : GetZero();
107     }
108     if (restrictions.Count == 2)
109     {
110         var value = restrictions[1];
111         if (AreEqual(index, any))
112         {
113             if (AreEqual(value, any))
114             {
115                 return Total; // Any - как отсутствие ограничения
116             }
117             return Add(SourcesTreeMethods.CountUsages(value),
    ↪ TargetsTreeMethods.CountUsages(value));
118         }
119         else
120         {
121             if (!Exists(index))
122             {
123                 return GetZero();
124             }
125             if (AreEqual(value, any))
126             {
127                 return GetOne();
128             }
129             ref var storedLinkValue = ref GetLinkReference(index);
130             if (AreEqual(storedLinkValue.Source, value) ||
    ↪ AreEqual(storedLinkValue.Target, value))
131             {
132                 return GetOne();
133             }
134             return GetZero();
135         }
136     }
137     if (restrictions.Count == 3)

```

```

138 {
139     var source = restrictions[constants.SourcePart];
140     var target = restrictions[constants.TargetPart];
141     if (AreEqual(index, any))
142     {
143         if (AreEqual(source, any) && AreEqual(target, any))
144         {
145             return Total;
146         }
147         else if (AreEqual(source, any))
148         {
149             return TargetsTreeMethods.CountUsages(target);
150         }
151         else if (AreEqual(target, any))
152         {
153             return SourcesTreeMethods.CountUsages(source);
154         }
155         else //if(source != Any && target != Any)
156         {
157             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
158             var link = SourcesTreeMethods.Search(source, target);
159             return AreEqual(link, constants.Null) ? GetZero() : GetOne();
160         }
161     }
162     else
163     {
164         if (!Exists(index))
165         {
166             return GetZero();
167         }
168         if (AreEqual(source, any) && AreEqual(target, any))
169         {
170             return GetOne();
171         }
172         ref var storedLinkValue = ref GetLinkReference(index);
173         if (!AreEqual(source, any) && !AreEqual(target, any))
174         {
175             if (AreEqual(storedLinkValue.Source, source) &&
176                 ⇨ AreEqual(storedLinkValue.Target, target))
177             {
178                 return GetOne();
179             }
180             return GetZero();
181         }
182         var value = default(TLink);
183         if (AreEqual(source, any))
184         {
185             value = target;
186         }
187         if (AreEqual(target, any))
188         {
189             value = source;
190         }
191         if (AreEqual(storedLinkValue.Source, value) ||
192             ⇨ AreEqual(storedLinkValue.Target, value))
193         {
194             return GetOne();
195         }
196         return GetZero();
197     }
198 }
199
200 [MethodImpl(MethodImplOptions.AggressiveInlining)]
201 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
202 {
203     var constants = Constants;
204     var @break = constants.Break;
205     if (restrictions.Count == 0)
206     {
207         for (var link = GetOne(); LessOrEqualThan(link,
208             ⇨ GetHeaderReference().AllocatedLinks); link = Increment(link))
209         {
210             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
211             {
212                 return @break;
213             }
214         }
215     }

```

```

212     }
213 }
214 return @break;
215 }
216 var @continue = constants.Continue;
217 var any = constants.Any;
218 var index = restrictions[constants.IndexPart];
219 if (restrictions.Count == 1)
220 {
221     if (AreEqual(index, any))
222     {
223         return Each(handler, Array.Empty<TLink>());
224     }
225     if (!Exists(index))
226     {
227         return @continue;
228     }
229     return handler(GetLinkStruct(index));
230 }
231 if (restrictions.Count == 2)
232 {
233     var value = restrictions[1];
234     if (AreEqual(index, any))
235     {
236         if (AreEqual(value, any))
237         {
238             return Each(handler, Array.Empty<TLink>());
239         }
240         if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
241         {
242             return @break;
243         }
244         return Each(handler, new Link<TLink>(index, any, value));
245     }
246     else
247     {
248         if (!Exists(index))
249         {
250             return @continue;
251         }
252         if (AreEqual(value, any))
253         {
254             return handler(GetLinkStruct(index));
255         }
256         ref var storedLinkValue = ref GetLinkReference(index);
257         if (AreEqual(storedLinkValue.Source, value) ||
258             AreEqual(storedLinkValue.Target, value))
259         {
260             return handler(GetLinkStruct(index));
261         }
262         return @continue;
263     }
264 }
265 if (restrictions.Count == 3)
266 {
267     var source = restrictions[constants.SourcePart];
268     var target = restrictions[constants.TargetPart];
269     if (AreEqual(index, any))
270     {
271         if (AreEqual(source, any) && AreEqual(target, any))
272         {
273             return Each(handler, Array.Empty<TLink>());
274         }
275         else if (AreEqual(source, any))
276         {
277             return TargetsTreeMethods.EachUsage(target, handler);
278         }
279         else if (AreEqual(target, any))
280         {
281             return SourcesTreeMethods.EachUsage(source, handler);
282         }
283         else //if(source != Any && target != Any)
284         {
285             var link = SourcesTreeMethods.Search(source, target);
286             return AreEqual(link, constants.Null) ? @continue :
287                 ↪ handler(GetLinkStruct(link));
288         }
289     }
290 }

```

```

289     else
290     {
291         if (!Exists(index))
292         {
293             return @continue;
294         }
295         if (AreEqual(source, any) && AreEqual(target, any))
296         {
297             return handler(GetLinkStruct(index));
298         }
299         ref var storedLinkValue = ref GetLinkReference(index);
300         if (!AreEqual(source, any) && !AreEqual(target, any))
301         {
302             if (AreEqual(storedLinkValue.Source, source) &&
303                 AreEqual(storedLinkValue.Target, target))
304             {
305                 return handler(GetLinkStruct(index));
306             }
307             return @continue;
308         }
309         var value = default(TLink);
310         if (AreEqual(source, any))
311         {
312             value = target;
313         }
314         if (AreEqual(target, any))
315         {
316             value = source;
317         }
318         if (AreEqual(storedLinkValue.Source, value) ||
319             AreEqual(storedLinkValue.Target, value))
320         {
321             return handler(GetLinkStruct(index));
322         }
323         return @continue;
324     }
325 }
326 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
327 }
328
329 /// <remarks>
330 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
331 ↳ в другом месте (но не в менеджере памяти, а в логике Links)
332 /// </remarks>
333 [MethodImpl(MethodImplOptions.AggressiveInlining)]
334 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
335 {
336     var constants = Constants;
337     var @null = constants.Null;
338     var linkIndex = restrictions[constants.IndexPart];
339     ref var link = ref GetLinkReference(linkIndex);
340     ref var header = ref GetHeaderReference();
341     ref var firstAsSource = ref header.FirstAsSource;
342     ref var firstAsTarget = ref header.FirstAsTarget;
343     // Будет корректно работать только в том случае, если пространство выделенной связи
344     ↳ предварительно заполнено нулями
345     if (!AreEqual(link.Source, @null))
346     {
347         SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
348     }
349     if (!AreEqual(link.Target, @null))
350     {
351         TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
352     }
353     link.Source = substitution[constants.SourcePart];
354     link.Target = substitution[constants.TargetPart];
355     if (!AreEqual(link.Source, @null))
356     {
357         SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
358     }
359     if (!AreEqual(link.Target, @null))
360     {
361         TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
362     }
363     return linkIndex;
364 }

```

```

364 /// <remarks>
365 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
366   ↳ пространство
367 /// </remarks>
368 [MethodImpl(MethodImplOptions.AggressiveInlining)]
369 public virtual TLink Create(ICollection<TLink> restrictions)
370 {
371     ref var header = ref GetHeaderReference();
372     var freeLink = header.FirstFreeLink;
373     if (!AreEqual(freeLink, Constants.Null))
374     {
375         UnusedLinksListMethods.Detach(freeLink);
376     }
377     else
378     {
379         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
380         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
381         {
382             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
383         }
384         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
385         {
386             _memory.ReservedCapacity += _memory.ReservationStep;
387             SetPointers(_memory);
388             header = ref GetHeaderReference();
389             header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
390                 ↳ LinkSizeInBytes);
391         }
392         header.AllocatedLinks = Increment(header.AllocatedLinks);
393         _memory.UsedCapacity += LinkSizeInBytes;
394         freeLink = header.AllocatedLinks;
395     }
396     return freeLink;
397 }
398
399 [MethodImpl(MethodImplOptions.AggressiveInlining)]
400 public virtual void Delete(ICollection<TLink> restrictions)
401 {
402     ref var header = ref GetHeaderReference();
403     var link = restrictions[Constants.IndexPart];
404     if (LessThan(link, header.AllocatedLinks)
405     {
406         UnusedLinksListMethods.AttachAsFirst(link);
407     }
408     else if (AreEqual(link, header.AllocatedLinks))
409     {
410         header.AllocatedLinks = Decrement(header.AllocatedLinks);
411         _memory.UsedCapacity -= LinkSizeInBytes;
412         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
413         // ↳ пока не дойдём до первой существующей связи
414         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
415         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
416             ↳ IsUnusedLink(header.AllocatedLinks))
417         {
418             UnusedLinksListMethods.Detach(header.AllocatedLinks);
419             header.AllocatedLinks = Decrement(header.AllocatedLinks);
420             _memory.UsedCapacity -= LinkSizeInBytes;
421         }
422     }
423 }
424
425 [MethodImpl(MethodImplOptions.AggressiveInlining)]
426 public IList<TLink> GetLinkStruct(TLink linkIndex)
427 {
428     ref var link = ref GetLinkReference(linkIndex);
429     return new Link<TLink>(linkIndex, link.Source, link.Target);
430 }
431
432 /// <remarks>
433 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
434   ↳ адрес реально поменялся
435 ///
436 /// Указатель this.links может быть в том же месте,
437 /// так как 0-я связь не используется и имеет такой же размер как Header,
438 /// поэтому header размещается в том же месте, что и 0-я связь
439 /// </remarks>
440 [MethodImpl(MethodImplOptions.AggressiveInlining)]
441 protected abstract void SetPointers(IResizableDirectMemory memory);

```

```

438 [MethodImpl(MethodImplOptions.AggressiveInlining)]
439 protected virtual void ResetPointers()
440 {
441     SourcesTreeMethods = null;
442     TargetsTreeMethods = null;
443     UnusedLinksListMethods = null;
444 }
445
446 [MethodImpl(MethodImplOptions.AggressiveInlining)]
447 protected abstract ref LinksHeader<TLink> GetHeaderReference();
448
449 [MethodImpl(MethodImplOptions.AggressiveInlining)]
450 protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
451
452 [MethodImpl(MethodImplOptions.AggressiveInlining)]
453 protected virtual bool Exists(TLink link)
454     => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
455     && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
456     && !IsUnusedLink(link);
457
458 [MethodImpl(MethodImplOptions.AggressiveInlining)]
459 protected virtual bool IsUnusedLink(TLink linkIndex)
460 {
461     if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
462         ↳ is not needed
463     {
464         ref var link = ref GetLinkReference(linkIndex);
465         return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
466     }
467     else
468     {
469         return true;
470     }
471 }
472
473 [MethodImpl(MethodImplOptions.AggressiveInlining)]
474 protected virtual TLink GetOne() => _one;
475
476 [MethodImpl(MethodImplOptions.AggressiveInlining)]
477 protected virtual TLink GetZero() => default;
478
479 [MethodImpl(MethodImplOptions.AggressiveInlining)]
480 protected virtual bool AreEqual(TLink first, TLink second) =>
481     ↳ _equalityComparer.Equals(first, second);
482
483 [MethodImpl(MethodImplOptions.AggressiveInlining)]
484 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
485     ↳ second) < 0;
486
487 [MethodImpl(MethodImplOptions.AggressiveInlining)]
488 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
489     ↳ _comparer.Compare(first, second) <= 0;
490
491 [MethodImpl(MethodImplOptions.AggressiveInlining)]
492 protected virtual bool GreaterThan(TLink first, TLink second) =>
493     ↳ _comparer.Compare(first, second) > 0;
494
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
497     ↳ _comparer.Compare(first, second) >= 0;
498
499 [MethodImpl(MethodImplOptions.AggressiveInlining)]
500 protected virtual long ConvertToInt64(TLink value) =>
501     ↳ _addressToInt64Converter.Convert(value);
502
503 [MethodImpl(MethodImplOptions.AggressiveInlining)]
504 protected virtual TLink ConvertToAddress(long value) =>
505     ↳ _int64ToAddressConverter.Convert(value);
506
507 [MethodImpl(MethodImplOptions.AggressiveInlining)]
508 protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
509     ↳ second);
510
511 [MethodImpl(MethodImplOptions.AggressiveInlining)]
512 protected virtual TLink Subtract(TLink first, TLink second) =>
513     ↳ Arithmetic<TLink>.Subtract(first, second);
514
515 [MethodImpl(MethodImplOptions.AggressiveInlining)]
516 protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);

```

```

507     [MethodImpl(MethodImplOptions.AggressiveInlining)]
508     protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
509
510     #region Disposable
511
512     protected override bool AllowMultipleDisposeCalls
513     {
514         [MethodImpl(MethodImplOptions.AggressiveInlining)]
515         get => true;
516     }
517
518     [MethodImpl(MethodImplOptions.AggressiveInlining)]
519     protected override void Dispose(bool manual, bool wasDisposed)
520     {
521         if (!wasDisposed)
522         {
523             ResetPointers();
524             _memory.DisposeIfPossible();
525         }
526     }
527
528     #endregion
529 }
530
531 }

```

#### 1.40 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/UnusedLinksListMethods.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Methods.Lists;
3  using Platform.Converters;
4  using static System.Runtime.CompilerServices.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
9  {
10     public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
11     ↪ ILinksListMethods<TLink>
12     {
13         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
14         ↪ UncheckedConverter<TLink, long>.Default;
15
16         private readonly byte* _links;
17         private readonly byte* _header;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnusedLinksListMethods(byte* links, byte* header)
21         {
22             _links = links;
23             _header = header;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
28         ↪ AsRef<LinksHeader<TLink>>(_header);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
32         ↪ AsRef<RawLink<TLink>>(_links + (RawLink<TLink>.SizeInBytes *
33         ↪ _addressToInt64Converter.Convert(link)));
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetSize() => GetHeaderReference().FreeLinks;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
52         ↪ element;
53     }
54 }

```

```

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

```

#### 1.41 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/ILinksListMethods.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.ResizableDirectMemory
6  {
7      public interface ILinksListMethods<TLink>
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         void Detach(TLink freeLink);
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         void AttachAsFirst(TLink link);
14     }
15 }

```

#### 1.42 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.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.ResizableDirectMemory
8  {
9      public interface ILinksTreeMethods<TLink>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         TLink CountUsages(TLink link);
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         TLink Search(TLink source, TLink target);
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         TLink EachUsage(TLink source, Func<IList<TLink>, TLink> handler);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         void Detach(ref TLink firstAsSource, TLink linkIndex);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         void Attach(ref TLink firstAsSource, TLink linkIndex);
25     }
26 }

```

#### 1.43 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.ResizableDirectMemory
9  {
10     public struct LinksHeader<TLink> : IEquatable<LinksHeader<TLink>>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
13
14         public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;
15
16         public TLink AllocatedLinks;

```



```

17     public TLink ReservedLinks;
18     public TLink FreeLinks;
19     public TLink FirstFreeLink;
20     public TLink FirstAsSource;
21     public TLink FirstAsTarget;
22     public TLink LastFreeLink;
23     public TLink Reserved8;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public override bool Equals(object obj) => obj is LinksHeader<TLink> linksHeader ?
        ↳ Equals(linksHeader) : false;
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     public bool Equals(LinksHeader<TLink> other)
30         => _equalityComparer.Equals(AllocatedLinks, other.AllocatedLinks)
31         && _equalityComparer.Equals(ReservedLinks, other.ReservedLinks)
32         && _equalityComparer.Equals(FreeLinks, other.FreeLinks)
33         && _equalityComparer.Equals(FirstFreeLink, other.FirstFreeLink)
34         && _equalityComparer.Equals(FirstAsSource, other.FirstAsSource)
35         && _equalityComparer.Equals(FirstAsTarget, other.FirstAsTarget)
36         && _equalityComparer.Equals(LastFreeLink, other.LastFreeLink)
37         && _equalityComparer.Equals(Reserved8, other.Reserved8);
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
        ↳ FirstFreeLink, FirstAsSource, FirstAsTarget, LastFreeLink, Reserved8).GetHashCode();
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
        ↳ left.Equals(right);
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
        ↳ !(left == right);
47 }
48 }

```

#### 1.44 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/RawLink.cs

```

1  using Platform.Unsafe;
2  using System;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.ResizableDirectMemory
9  {
10     public struct RawLink<TLink> : IEquatable<RawLink<TLink>>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
13
14         public static readonly long SizeInBytes = Structure<RawLink<TLink>>.Size;
15
16         public TLink Source;
17         public TLink Target;
18         public TLink LeftAsSource;
19         public TLink RightAsSource;
20         public TLink SizeAsSource;
21         public TLink LeftAsTarget;
22         public TLink RightAsTarget;
23         public TLink SizeAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public override bool Equals(object obj) => obj is RawLink<TLink> link ? Equals(link) :
            ↳ false;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public bool Equals(RawLink<TLink> other)
30             => _equalityComparer.Equals(Source, other.Source)
31             && _equalityComparer.Equals(Target, other.Target)
32             && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
33             && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
34             && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
35             && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
36             && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
37             && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

40     public override int GetHashCode() => (Source, Target, LeftAsSource, RightAsSource,
    ↪ SizeAsSource, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public static bool operator ==(RawLink<TLink> left, RawLink<TLink> right) =>
    ↪ left.Equals(right);
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public static bool operator !=(RawLink<TLink> left, RawLink<TLink> right) => !(left ==
    ↪ right);
47 }
48 }

```

#### 1.45 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksAvlBalancedTreeMethodsBase

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.ResizableDirectMemory.Generic;
3  using static System.Runtime.CompilerServices.Unsafe;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
8  {
9      public unsafe abstract class UInt64LinksAvlBalancedTreeMethodsBase :
    ↪ LinksAvlBalancedTreeMethodsBase<ulong>
10     {
11         protected new readonly RawLink<ulong>* Links;
12         protected new readonly LinksHeader<ulong>* Header;
13
14         protected UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
    ↪ RawLink<ulong>* links, LinksHeader<ulong>* header)
15             : base(constants, (byte*)links, (byte*)header)
16         {
17             Links = links;
18             Header = header;
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ulong GetZero() => 0UL;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override bool EqualToZero(ulong value) => value == 0UL;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override bool AreEqual(ulong first, ulong second) => first == second;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override bool GreaterThanZero(ulong value) => value > 0UL;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override bool GreaterThan(ulong first, ulong second) => first > second;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
    ↪ always true for ulong
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
    ↪ always >= 0 for ulong
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
    ↪ for ulong
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override bool LessThan(ulong first, ulong second) => first < second;
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected override ulong Increment(ulong value) => ++value;
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected override ulong Decrement(ulong value) => --value;
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override ulong Add(ulong first, ulong second) => first + second;

```

```

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,
71         ↪ secondLink.Source, secondLink.Target);
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
76 {
77     ref var firstLink = ref Links[first];
78     ref var secondLink = ref Links[second];
79     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
80         ↪ secondLink.Source, secondLink.Target);
81 }
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected override ulong GetSizeValue(ulong value) => (value & 4294967264UL) >> 5;
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
88     ↪ storedValue & 31UL | (size & 134217727UL) << 5;
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 protected override bool GetLeftIsChildValue(ulong value) => (value & 16UL) >> 4 == 1UL;
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
95     ↪ storedValue = storedValue & 4294967279UL | (As<bool, byte>(ref value) & 1UL) << 4;
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 protected override bool GetRightIsChildValue(ulong value) => (value & 8UL) >> 3 == 1UL;
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
102     ↪ storedValue = storedValue & 4294967287UL | (As<bool, byte>(ref value) & 1UL) << 3;
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
106     ↪ 0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
107     ↪ sbyte
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
111     ↪ storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
112     ↪ value & 3) & 7UL);
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
119 }
120 }

```

#### 1.46 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSizeBalancedTreeMethodsBase

```

1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.ResizableDirectMemory.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
7 {
8     public unsafe abstract class UInt64LinksSizeBalancedTreeMethodsBase :
9         ↪ LinksSizeBalancedTreeMethodsBase<ulong>
10     {
11         protected new readonly RawLink<ulong>* Links;
12         protected new readonly LinksHeader<ulong>* Header;
13
14         protected UInt64LinksSizeBalancedTreeMethodsBase(LinksConstants<ulong> constants,
15             ↪ RawLink<ulong>* links, LinksHeader<ulong>* header)
16             : base(constants, (byte*)links, (byte*)header)
17         {
18
19         }
20     }
21 }

```

```

16     Links = links;
17     Header = header;
18 }
19
20 [MethodImpl(MethodImplOptions.AggressiveInlining)]
21 protected override ulong GetZero() => OUL;
22
23 [MethodImpl(MethodImplOptions.AggressiveInlining)]
24 protected override bool EqualToZero(ulong value) => value == OUL;
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 > OUL;
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 == OUL; // value is
    ↳ always >= 0 for ulong
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
    ↳ for ulong
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected override bool LessThan(ulong first, ulong second) => first < second;
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override ulong Increment(ulong value) => ++value;
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override ulong Decrement(ulong value) => --value;
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected override ulong Add(ulong first, ulong second) => first + second;
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override ulong Subtract(ulong first, ulong second) => first - second;
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
67 {
68     ref var firstLink = ref Links[first];
69     ref var secondLink = ref Links[second];
70     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↳ secondLink.Source, secondLink.Target);
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
75 {
76     ref var firstLink = ref Links[first];
77     ref var secondLink = ref Links[second];
78     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↳ secondLink.Source, secondLink.Target);
79 }
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
86 }
87 }

```

1.47 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesAvlBalancedTreeMeth

```
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.ResizableDirectMemory.Specific
6  {
7      public unsafe class UInt64LinksSourcesAvlBalancedTreeMethods :
8          ↳ UInt64LinksAvlBalancedTreeMethodsBase
9      {
10
11          public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
12              ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
13              ↳ { }
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected override ref ulong GetLeftReference(ulong node) => ref
17              ↳ Links[node].LeftAsSource;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override ref ulong GetRightReference(ulong node) => ref
21              ↳ Links[node].RightAsSource;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
31              ↳ left;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
35              ↳ right;
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
42              ↳ Links[node].SizeAsSource, size);
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          protected override bool GetLeftIsChild(ulong node) =>
46              ↳ GetLeftIsChildValue(Links[node].SizeAsSource);
47
48          //[MethodImpl(MethodImplOptions.AggressiveInlining)]
49          //protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void SetLeftIsChild(ulong node, bool value) =>
53              ↳ SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
54
55          [MethodImpl(MethodImplOptions.AggressiveInlining)]
56          protected override bool GetRightIsChild(ulong node) =>
57              ↳ GetRightIsChildValue(Links[node].SizeAsSource);
58
59          //[MethodImpl(MethodImplOptions.AggressiveInlining)]
60          //protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          protected override void SetRightIsChild(ulong node, bool value) =>
64              ↳ SetRightIsChildValue(ref Links[node].SizeAsSource, value);
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected override sbyte GetBalance(ulong node) =>
68              ↳ GetBalanceValue(Links[node].SizeAsSource);
69
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
72              ↳ Links[node].SizeAsSource, value);
73
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          protected override ulong GetTreeRoot() => Header->FirstAsSource;
76
77          [MethodImpl(MethodImplOptions.AggressiveInlining)]
78          protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
79
80      }
81  }
```

```

65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override bool FirstIsToLeftOfSecond(ulong firstSource, ulong firstTarget,
67     ↪ ulong secondSource, ulong secondTarget)
68     => firstSource < secondSource || (firstSource == secondSource && firstTarget <
69     ↪ secondTarget);
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
73     ↪ ulong secondSource, ulong secondTarget)
74     => firstSource > secondSource || (firstSource == secondSource && firstTarget >
75     ↪ secondTarget);
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected override void ClearNode(ulong node)
79 {
80     ref var link = ref Links[node];
81     link.LeftAsSource = OUL;
82     link.RightAsSource = OUL;
83     link.SizeAsSource = OUL;
84 }

```

#### 1.48 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesSizeBalancedTreeMet

```

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.ResizableDirectMemory.Specific
6 {
7     public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
8     ↪ UInt64LinksSizeBalancedTreeMethodsBase
9     {
10         public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↪ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↪ Links[node].LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↪ Links[node].RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
30             ↪ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
34             ↪ right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
41             ↪ size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override ulong GetTreeRoot() => Header->FirstAsSource;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool FirstIsToLeftOfSecond(ulong firstSource, ulong firstTarget,
51             ↪ ulong secondSource, ulong secondTarget)
52             => firstSource < secondSource || (firstSource == secondSource && firstTarget <
53             ↪ secondTarget);
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

1.49 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsAvlBalancedTreeMeth

```

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.ResizableDirectMemory.Specific
6  {
7      public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
8          ↪ UInt64LinksAvlBalancedTreeMethodsBase
9      {
10         public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↪ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↪ Links[node].LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↪ Links[node].RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
30             ↪ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
34             ↪ right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsTarget);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
41             ↪ Links[node].SizeAsTarget, size);
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override bool GetLeftIsChild(ulong node) =>
45             ↪ GetLeftIsChildValue(Links[node].SizeAsTarget);
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override void SetLeftIsChild(ulong node, bool value) =>
49             ↪ SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override bool GetRightIsChild(ulong node) =>
53             ↪ GetRightIsChildValue(Links[node].SizeAsTarget);
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override void SetRightIsChild(ulong node, bool value) =>
57             ↪ SetRightIsChildValue(ref Links[node].SizeAsTarget, value);
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override sbyte GetBalance(ulong node) =>
61             ↪ GetBalanceValue(Links[node].SizeAsTarget);
62     }
63 }

```

```

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->FirstAsTarget;
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 = OUL;
72         link.RightAsTarget = OUL;
73         link.SizeAsTarget = OUL;
74     }
75 }
76 }

```

## 1.50 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMet

```

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.ResizableDirectMemory.Specific
6  {
7      public unsafe class UInt64LinksTargetsSizeBalancedTreeMethods :
    ↳ UInt64LinksSizeBalancedTreeMethodsBase
8      {
9          public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
    ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
    ↳ { }
10
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected override ref ulong GetLeftReference(ulong node) => ref
    ↳ Links[node].LeftAsTarget;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetRightReference(ulong node) => ref
    ↳ Links[node].RightAsTarget;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
    ↳ left;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
    ↳ right;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
    ↳ size;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override ulong GetTreeRoot() => Header->FirstAsTarget;
37

```



```

38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
43     ↪ ulong secondSource, ulong secondTarget)
44     => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
45     ↪ secondSource);
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
49     ↪ ulong secondSource, ulong secondTarget)
50     => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
51     ↪ secondSource);
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override void ClearNode(ulong node)
55 {
56     ref var link = ref Links[node];
57     link.LeftAsTarget = OUL;
58     link.RightAsTarget = OUL;
59     link.SizeAsTarget = OUL;
60 }
61
62 }
63
64 }

```

1.51 `./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64ResizableDirectMemoryLinks.cs`

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using Platform.Memory;
4 using Platform.Data.Doublets.ResizableDirectMemory.Generic;
5 using Platform.Singletons;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
10 {
11     /// <summary>
12     /// <para>Represents a low-level implementation of direct access to resizable memory, for
13     ///     ↪ organizing the storage of links with addresses represented as <see cref="System.UInt64"
14     ///     ↪ >/>.</para>
15     /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
16     ///     ↪ размером, для организации хранения связей с адресами представленными в виде <see
17     ///     ↪ cref="System.UInt64"/>.</para>
18     /// </summary>
19     public unsafe class UInt64ResizableDirectMemoryLinks : ResizableDirectMemoryLinksBase<ulong>
20     {
21         private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
22         private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
23         private LinksHeader<ulong>* _header;
24         private RawLink<ulong>* _links;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public UInt64ResizableDirectMemoryLinks(string address) : this(address,
28             ↪ DefaultLinksSizeStep) { }
29
30         /// <summary>
31         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
32         ///     ↪ минимальным шагом расширения базы данных.
33         /// </summary>
34         /// <param name="address">Полный путь к файлу базы данных.</param>
35         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
36         ///     ↪ байтах.</param>
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
39             ↪ this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
40             ↪ memoryReservationStep) { }
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
44             ↪ DefaultLinksSizeStep) { }
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
48             ↪ memoryReservationStep) : this(memory, memoryReservationStep,
49             ↪ Default<LinksConstants<ulong>>.Instance, true) { }
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     }
53 }

```

```

40 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep, LinksConstants<ulong> constants, bool useAvlBasedIndex) :
    ↳ base(memory, memoryReservationStep, constants)
41 {
42     if (useAvlBasedIndex)
43     {
44         _createSourceTreeMethods = () => new
            ↳ UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
45         _createTargetTreeMethods = () => new
            ↳ UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
46     }
47     else
48     {
49         _createSourceTreeMethods = () => new
            ↳ UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
50         _createTargetTreeMethods = () => new
            ↳ UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
51     }
52     Init(memory, memoryReservationStep);
53 }
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 protected override void SetPointers(IResizableDirectMemory memory)
57 {
58     _header = (LinksHeader<ulong>*)memory.Pointer;
59     _links = (RawLink<ulong>*)memory.Pointer;
60     SourcesTreeMethods = _createSourceTreeMethods();
61     TargetsTreeMethods = _createTargetTreeMethods();
62     UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
63 }
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override void ResetPointers()
67 {
68     base.ResetPointers();
69     _links = null;
70     _header = null;
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
    ↳ _links[linkIndex];
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 protected override bool AreEqual(ulong first, ulong second) => first == second;
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected override bool LessThan(ulong first, ulong second) => first < second;
84
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 protected override bool GreaterThan(ulong first, ulong second) => first > second;
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected override ulong GetZero() => 0UL;
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 protected override ulong GetOne() => 1UL;
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 protected override long ConvertToInt64(ulong value) => (long)value;
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected override ulong ConvertToAddress(long value) => (ulong)value;
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 protected override ulong Add(ulong first, ulong second) => first + second;
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 protected override ulong Subtract(ulong first, ulong second) => first - second;
111

```

```

112     [MethodImpl(MethodImplOptions.AggressiveInlining)]
113     protected override ulong Increment(ulong link) => ++link;
114
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     protected override ulong Decrement(ulong link) => --link;
117 }
118 }

```

### 1.52 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64UnusedLinksListMethods.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.ResizableDirectMemory.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
7 {
8     public unsafe class UInt64UnusedLinksListMethods : UnusedLinksListMethods<ulong>
9     {
10         private readonly RawLink<ulong>* _links;
11         private readonly LinksHeader<ulong>* _header;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public UInt64UnusedLinksListMethods(RawLink<ulong>* links, LinksHeader<ulong>* header)
15             : base((byte*)links, (byte*)header)
16         {
17             _links = links;
18             _header = header;
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref _links[link];
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
26     }
27 }

```

### 1.53 ./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.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.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(ICollection<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
29                 // ↪ generics) but will be possible with Sigil
30                 var halvedSequence = new TLink[(length / 2) + (length % 2)];
31                 HalveSequence(halvedSequence, sequence, length);
32                 sequence = halvedSequence;
33                 length = halvedSequence.Length;
34             }
35             // Keep creating layer after layer
36             while (length > 2)
37             {
38                 HalveSequence(sequence, sequence, length);
39                 length = (length / 2) + (length % 2);
40             }
41             return _links.GetOrCreate(sequence[0], sequence[1]);
42         }
43     }
44 }

```

```

41     }
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     private void HalveSequence(ICollection<TLink> destination, ICollection<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 }

```

#### 1.54 ./csharp/Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs

```

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
16     ///     ↳ Links на этапе сжатия.
17     ///     А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
18     ///     ↳ таком случае тип значения элемента массива может быть любым, как char так и ulong.
19     ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
20     ///     ↳ пар, а так же разом выполнить замену.
21     /// </remarks>
22     public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
23     {
24         private static readonly LinksConstants<TLink> _constants =
25             ↳ Default<LinksConstants<TLink>>.Instance;
26         private static readonly EqualityComparer<TLink> _equalityComparer =
27             ↳ EqualityComparer<TLink>.Default;
28         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
29
30         private static readonly TLink _zero = default;
31         private static readonly TLink _one = Arithmetic.Increment(_zero);
32
33         private readonly IConverter<ICollection<TLink>, TLink> _baseConverter;
34         private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
35         private readonly TLink _minFrequencyToCompress;
36         private readonly bool _doInitialFrequenciesIncrement;
37         private Doublet<TLink> _maxDoublet;
38         private LinkFrequency<TLink> _maxDoubletData;
39
40         private struct HalfDoublet
41         {
42             public TLink Element;
43             public LinkFrequency<TLink> DoubletData;
44
45             [MethodImpl(MethodImplOptions.AggressiveInlining)]
46             public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
47             {
48                 Element = element;
49                 DoubletData = doubletData;
50             }
51
52             public override string ToString() => $"{Element}: ({DoubletData})";
53         }
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public CompressingConverter(ICollection<TLink> links, IConverter<ICollection<TLink>, TLink>
57             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
58             : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

55 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
    ↳ doInitialFrequenciesIncrement)
56 : this(links, baseConverter, doubletFrequenciesCache, _one,
    ↳ doInitialFrequenciesIncrement) { }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
    ↳ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
60 : base(links)
61 {
62     _baseConverter = baseConverter;
63     _doubletFrequenciesCache = doubletFrequenciesCache;
64     if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
65     {
66         minFrequencyToCompress = _one;
67     }
68     _minFrequencyToCompress = minFrequencyToCompress;
69     _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
70     ResetMaxDoublet();
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public override TLink Convert(IList<TLink> source) =>
    ↳ _baseConverter.Convert(Compress(source));
75
76 /// <remarks>
77 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
78 /// Faster version (doublets' frequencies dictionary is not recreated).
79 /// </remarks>
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 private IList<TLink> Compress(IList<TLink> sequence)
82 {
83     if (sequence.IsNullOrEmpty())
84     {
85         return null;
86     }
87     if (sequence.Count == 1)
88     {
89         return sequence;
90     }
91     if (sequence.Count == 2)
92     {
93         return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
94     }
95     // TODO: arraypool with min size (to improve cache locality) or stackalloc 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.Source = sequence[i - 1];
101         doublet.Target = sequence[i];
102         LinkFrequency<TLink> data;
103         if (_doInitialFrequenciesIncrement)
104         {
105             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
106         }
107         else
108         {
109             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
110             if (data == null)
111             {
112                 throw new NotSupportedException("If you ask not to increment
                    ↳ frequencies, it is expected that all frequencies for the sequence
                    ↳ are prepared.");
113             }
114         }
115         copy[i - 1].Element = sequence[i - 1];
116         copy[i - 1].DoubletData = data;
117         UpdateMaxDoublet(ref doublet, data);
118     }
119     copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
120     copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
121     if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
122     {
123         var newLength = ReplaceDoublets(copy);
124         sequence = new TLink[newLength];
    
```

```

125         for (int i = 0; i < newLength; i++)
126         {
127             sequence[i] = copy[i].Element;
128         }
129     }
130     return sequence;
131 }
132
133 /// <remarks>
134 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
135 /// </remarks>
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 private int ReplaceDoublets(HalfDoublet[] copy)
138 {
139     var oldLength = copy.Length;
140     var newLength = copy.Length;
141     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
142     {
143         var maxDoubletSource = _maxDoublet.Source;
144         var maxDoubletTarget = _maxDoublet.Target;
145         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
146         {
147             _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
148                 ↪ maxDoubletTarget);
149         }
150         var maxDoubletReplacementLink = _maxDoubletData.Link;
151         oldLength--;
152         var oldLengthMinusTwo = oldLength - 1;
153         // Substitute all usages
154         int w = 0, r = 0; // (r == read, w == write)
155         for (; r < oldLength; r++)
156         {
157             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
158                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
159             {
160                 if (r > 0)
161                 {
162                     var previous = copy[w - 1].Element;
163                     copy[w - 1].DoubletData.DecrementFrequency();
164                     copy[w - 1].DoubletData =
165                         ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
166                             ↪ maxDoubletReplacementLink);
167                 }
168                 if (r < oldLengthMinusTwo)
169                 {
170                     var next = copy[r + 2].Element;
171                     copy[r + 1].DoubletData.DecrementFrequency();
172                     copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
173                         ↪ next);
174                 }
175                 copy[w++] = maxDoubletReplacementLink;
176                 r++;
177                 newLength--;
178             }
179             else
180             {
181                 copy[w++] = copy[r];
182             }
183         }
184         oldLength = newLength;
185         ResetMaxDoublet();
186         UpdateMaxDoublet(copy, newLength);
187     }
188     return newLength;
189 }
190
191 [MethodImpl(MethodImplOptions.AggressiveInlining)]
192 private void ResetMaxDoublet()
193 {
194     _maxDoublet = new Doublet<TLink>();
195     _maxDoubletData = new LinkFrequency<TLink>();
196 }
197
198 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

198 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
199 {
200     Doublet<TLink> doublet = default;
201     for (var i = 1; i < length; i++)
202     {
203         doublet.Source = copy[i - 1].Element;
204         doublet.Target = copy[i].Element;
205         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
206     }
207 }
208
209 [MethodImpl(MethodImplOptions.AggressiveInlining)]
210 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
211 {
212     var frequency = data.Frequency;
213     var maxFrequency = _maxDoubletData.Frequency;
214     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
215     ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
216     ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
217     ↪ _maxDoublet.Target)))
218     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
219     ↪ (_comparer.Compare(maxFrequency, frequency) < 0 ||
220     ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
221     ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
222     ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
223     ↪ better stability and better compression on sequent data and even on random
224     ↪ numbers data (but gives collisions anyway) */
225     {
226         _maxDoublet = doublet;
227         _maxDoubletData = data;
228     }
229 }
230 }
231 }
232 }
233 }

```

#### 1.55 ./csharp/Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs

```

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>,
10     ↪ IConverter<IList<TLink>, TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         protected LinksListToSequenceConverterBase(ILinks<TLink> links) : base(links) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public abstract TLink Convert(IList<TLink> source);
17     }
18 }

```

#### 1.56 ./csharp/Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs

```

1 using System.Collections.Generic;
2 using System.Linq;
3 using System.Runtime.CompilerServices;
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.Converters
9 {
10     public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13         ↪ EqualityComparer<TLink>.Default;
14         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
15
16         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
20         ↪ sequenceToItsLocalElementLevelsConverter) : base(links)
21         => _sequenceToItsLocalElementLevelsConverter =
22         ↪ sequenceToItsLocalElementLevelsConverter;
23     }
24 }

```

```

21 [MethodImpl(MethodImplOptions.AggressiveInlining)]
22 public override TLink Convert(TList<TLink> sequence)
23 {
24     var length = sequence.Count;
25     if (length == 1)
26     {
27         return sequence[0];
28     }
29     if (length == 2)
30     {
31         return _links.GetOrCreate(sequence[0], sequence[1]);
32     }
33     sequence = sequence.ToArray();
34     var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
35     while (length > 2)
36     {
37         var levelRepeat = 1;
38         var currentLevel = levels[0];
39         var previousLevel = levels[0];
40         var skipOnce = false;
41         var w = 0;
42         for (var i = 1; i < length; i++)
43         {
44             if (_equalityComparer.Equals(currentLevel, levels[i]))
45             {
46                 levelRepeat++;
47                 skipOnce = false;
48                 if (levelRepeat == 2)
49                 {
50                     sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
51                     var newLevel = i >= length - 1 ?
52                         GetPreviousLowerThanCurrentOrCurrent(previousLevel,
53                             ↪ currentLevel) :
54                         i < 2 ?
55                         GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
56                         GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
57                             ↪ currentLevel, levels[i + 1]);
58                     levels[w] = newLevel;
59                     previousLevel = currentLevel;
60                     w++;
61                     levelRepeat = 0;
62                     skipOnce = true;
63                 }
64                 else if (i == length - 1)
65                 {
66                     sequence[w] = sequence[i];
67                     levels[w] = levels[i];
68                     w++;
69                 }
70             }
71             else
72             {
73                 currentLevel = levels[i];
74                 levelRepeat = 1;
75                 if (skipOnce)
76                 {
77                     skipOnce = false;
78                 }
79                 else
80                 {
81                     sequence[w] = sequence[i - 1];
82                     levels[w] = levels[i - 1];
83                     previousLevel = levels[w];
84                     w++;
85                 }
86                 if (i == length - 1)
87                 {
88                     sequence[w] = sequence[i];
89                     levels[w] = levels[i];
90                     w++;
91                 }
92             }
93         }
94         length = w;
95     }
96     return _links.GetOrCreate(sequence[0], sequence[1]);
97 }
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

98     private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
    ↪     current, TLink next)
99     {
100         return _comparer.Compare(previous, next) > 0
101             ? _comparer.Compare(previous, current) < 0 ? previous : current
102             : _comparer.Compare(next, current) < 0 ? next : current;
103     }
104
105     [MethodImpl(MethodImplOptions.AggressiveInlining)]
106     private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
    ↪     _comparer.Compare(next, current) < 0 ? next : current;
107
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
    ↪     => _comparer.Compare(previous, current) < 0 ? previous : current;
110 }
111 }

```

## 1.57 ./csharp/Platform.Data.Doublets/Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs

```

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 class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
    ↪      IConverter<ILink<TLink>>
10     {
11         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
    ↪         IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
    ↪         => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public ILink<TLink> Convert(ILink<TLink> sequence)
20         {
21             var levels = new TLink[sequence.Count];
22             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
23             for (var i = 1; i < sequence.Count - 1; i++)
24             {
25                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
26                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
27                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
28             }
29             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
    ↪             sequence[sequence.Count - 1]);
30             return levels;
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public TLink GetFrequencyNumber(TLink source, TLink target) =>
    ↪         _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
35     }
36 }

```

## 1.58 ./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs

```

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.CriterionMatchers
7  {
8      public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
    ↪      ICriterionMatcher<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public bool IsMatched(TLink argument) => _links.IsPartialPoint(argument);
15     }
16 }

```

### 1.59 ./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.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.Sequences.CriterionMatchers
8 {
9     public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↳ EqualityComparer<TLink>.Default;
13
14         private readonly ILinks<TLink> _links;
15         private readonly TLink _sequenceMarkerLink;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
19         {
20             _links = links;
21             _sequenceMarkerLink = sequenceMarkerLink;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public bool IsMatched(TLink sequenceCandidate)
26         => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
27             || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
28                 ↳ sequenceCandidate), _links.Constants.Null);
29     }
30 }

```

### 1.60 ./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>,
12         ↳ ISequenceAppender<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↳ EqualityComparer<TLink>.Default;
16
17         private readonly IStack<TLink> _stack;
18         private readonly ISequenceHeightProvider<TLink> _heightProvider;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
22             ↳ ISequenceHeightProvider<TLink> heightProvider)
23             : base(links)
24         {
25             _stack = stack;
26             _heightProvider = heightProvider;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public TLink Append(TLink sequence, TLink appendant)
31         {
32             var cursor = sequence;
33             var links = _links;
34             while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
35             {
36                 var source = links.GetSource(cursor);
37                 var target = links.GetTarget(cursor);
38                 if (_equalityComparer.Equals(_heightProvider.Get(source),
39                     ↳ _heightProvider.Get(target)))
40                 {
41                     break;
42                 }
43                 else
44                 {
45                     _stack.Push(source);
46                     cursor = target;
47                 }
48             }
49         }
50     }
51 }

```

```

44     }
45     var left = cursor;
46     var right = appendant;
47     while (!_equalityComparer.Equals(cursor = _stack.Pop(), links.Constants.Null))
48     {
49         right = links.GetOrCreate(left, right);
50         left = cursor;
51     }
52     return links.GetOrCreate(left, right);
53 }
54 }
55 }

```

#### 1.61 ./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
8  namespace Platform.Data.Doublets.Sequences
9  {
10     public class DuplicateSegmentsCounter<TLink> : ICounter<int>
11     {
12         private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
13             ↪ _duplicateFragmentsProvider;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
17             ↪ IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
18             ↪ duplicateFragmentsProvider;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
22     }
23 }

```

#### 1.62 ./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Interfaces;
6  using Platform.Collections;
7  using Platform.Collections.Lists;
8  using Platform.Collections.Segments;
9  using Platform.Collections.Segments.Walkers;
10 using Platform.Singletons;
11 using Platform.Converters;
12 using Platform.Data.Doublets.Unicode;
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 DuplicateSegmentsProvider<TLink> :
19         ↪ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
20         ↪ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
21     {
22         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
23             ↪ UncheckedConverter<TLink, long>.Default;
24         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
25             ↪ UncheckedConverter<TLink, ulong>.Default;
26         private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
27             ↪ UncheckedConverter<ulong, TLink>.Default;
28
29         private readonly ILinks<TLink> _links;
30         private readonly ILinks<TLink> _sequences;
31         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
32         private BitString _visited;
33
34         private class ItemEqualityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
35             ↪ IList<TLink>>>
36         {
37             private readonly IListEqualityComparer<TLink> _listComparer;
38
39             public ItemEqualityComparer() => _listComparer =
40                 ↪ Default<IListEqualityComparer<TLink>>.Instance;
41
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

36     public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
37         ↪ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
38         ↪ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
39         ↪ right.Value);
40
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
44         ↪ (_listComparer.GetHashCode(pair.Key),
45         ↪ _listComparer.GetHashCode(pair.Value)).GetHashCode();
46
47 }
48
49 private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
50 {
51     private readonly IListComparer<TLink> _listComparer;
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
58         ↪ KeyValuePair<IList<TLink>, IList<TLink>> right)
59     {
60         var intermediateResult = _listComparer.Compare(left.Key, right.Key);
61         if (intermediateResult == 0)
62         {
63             intermediateResult = _listComparer.Compare(left.Value, right.Value);
64         }
65         return intermediateResult;
66     }
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
71     : base(minimumStringSegmentLength: 2)
72 {
73     _links = links;
74     _sequences = sequences;
75 }
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
79 {
80     _groups = new HashSet<KeyValuePair<IList<TLink>,
81         ↪ IList<TLink>>>(Default<ItemEqualityComparer>.Instance);
82     var links = _links;
83     var count = links.Count();
84     _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
85     links.Each(link =>
86     {
87         var linkIndex = links.GetIndex(link);
88         var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
89         var constants = links.Constants;
90         if (!_visited.Get(linkBitIndex))
91         {
92             var sequenceElements = new List<TLink>();
93             var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
94             _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
95                 ↪ LinkAddress<TLink>(linkIndex));
96             if (sequenceElements.Count > 2)
97             {
98                 WalkAll(sequenceElements);
99             }
100         }
101         return constants.Continue;
102     });
103     var resultList = _groups.ToList();
104     var comparer = Default<ItemComparer>.Instance;
105     resultList.Sort(comparer);
106
107     #if DEBUG
108     foreach (var item in resultList)
109     {
110         PrintDuplicates(item);
111     }
112     #endif
113     return resultList;
114 }
115
116 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

106     protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
107         ↪ length) => new Segment<TLink>(elements, offset, length);
108
109     [MethodImpl(MethodImplOptions.AggressiveInlining)]
110     protected override void OnDuplicateFound(Segment<TLink> segment)
111     {
112         var duplicates = CollectDuplicatesForSegment(segment);
113         if (duplicates.Count > 1)
114         {
115             _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
116                 ↪ duplicates));
117         }
118     }
119
120     [MethodImpl(MethodImplOptions.AggressiveInlining)]
121     private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
122     {
123         var duplicates = new List<TLink>();
124         var readAsElement = new HashSet<TLink>();
125         var restrictions = segment.ShiftRight();
126         var constants = _links.Constants;
127         restrictions[0] = constants.Any;
128         _sequences.Each(sequence =>
129         {
130             var sequenceIndex = sequence[constants.IndexPart];
131             duplicates.Add(sequenceIndex);
132             readAsElement.Add(sequenceIndex);
133             return constants.Continue;
134         }, restrictions);
135         if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
136         {
137             return new List<TLink>();
138         }
139         foreach (var duplicate in duplicates)
140         {
141             var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
142             _visited.Set(duplicateBitIndex);
143         }
144         if (_sequences is Sequences sequencesExperiments)
145         {
146             var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
147                 ↪ ashSet<ulong>)(object)readAsElement,
148                 ↪ (IList<ulong>)segment);
149             foreach (var partiallyMatchedSequence in partiallyMatched)
150             {
151                 var sequenceIndex =
152                     ↪ _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
153                 duplicates.Add(sequenceIndex);
154             }
155         }
156         duplicates.Sort();
157         return duplicates;
158     }
159
160     [MethodImpl(MethodImplOptions.AggressiveInlining)]
161     private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
162     {
163         if (!(_links is ILinks<ulong> ulongLinks))
164         {
165             return;
166         }
167         var duplicatesKey = duplicatesItem.Key;
168         var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
169         Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)}");
170         var duplicatesList = duplicatesItem.Value;
171         for (int i = 0; i < duplicatesList.Count; i++)
172         {
173             var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
174             var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
175                 ↪ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
176                 ↪ UnicodeMap.IsCharLink(link.Index) ?
177                 ↪ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
178             Console.WriteLine(formattedSequenceStructure);
179             var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
180                 ↪ ulongLinks);
181             Console.WriteLine(sequenceString);
182         }
183         Console.WriteLine();
184     }

```

```

175     }
176 }
177 }

```

### 1.63 ./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
13     /// ↪ between them).
14     /// TODO: Extract interface to implement frequencies storage inside Links storage
15     /// </remarks>
16     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
17     {
18         private static readonly EqualityComparer<TLink> _equalityComparer =
19             ↪ EqualityComparer<TLink>.Default;
20         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
21
22         private static readonly TLink _zero = default;
23         private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
26         private readonly ICounter<TLink, TLink> _frequencyCounter;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
30             : base(links)
31         {
32             _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
33                 ↪ DoubletComparer<TLink>.Default);
34             _frequencyCounter = frequencyCounter;
35         }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
39         {
40             var doublet = new Doublet<TLink>(source, target);
41             return GetFrequency(ref doublet);
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
46         {
47             _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
48             return data;
49         }
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         public void IncrementFrequencies(IList<TLink> sequence)
53         {
54             for (var i = 1; i < sequence.Count; i++)
55             {
56                 IncrementFrequency(sequence[i - 1], sequence[i]);
57             }
58         }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
62         {
63             var doublet = new Doublet<TLink>(source, target);
64             return IncrementFrequency(ref doublet);
65         }
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public void PrintFrequencies(IList<TLink> sequence)
69         {
70             for (var i = 1; i < sequence.Count; i++)
71             {
72                 PrintFrequency(sequence[i - 1], sequence[i]);
73             }
74         }
75     }
76 }

```

```

73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public void PrintFrequency(TLink source, TLink target)
75 {
76     var number = GetFrequency(source, target).Frequency;
77     Console.WriteLine("{0},{1} - {2}", source, target, number);
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
82 {
83     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
84     {
85         data.IncrementFrequency();
86     }
87     else
88     {
89         var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
90         data = new LinkFrequency<TLink>(_one, link);
91         if (!_equalityComparer.Equals(link, default))
92         {
93             data.Frequency = Arithmetic.Add(data.Frequency,
94                 ↪ _frequencyCounter.Count(link));
95         }
96         _doubletsCache.Add(doublet, data);
97     }
98     return data;
99 }
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public void ValidateFrequencies()
102 {
103     foreach (var entry in _doubletsCache)
104     {
105         var value = entry.Value;
106         var linkIndex = value.Link;
107         if (!_equalityComparer.Equals(linkIndex, default))
108         {
109             var frequency = value.Frequency;
110             var count = _frequencyCounter.Count(linkIndex);
111             // TODO: Why `frequency` always greater than `count` by 1?
112             if (((_comparer.Compare(frequency, count) > 0) &&
113                 ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
114                 || ((_comparer.Compare(count, frequency) > 0) &&
115                 ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
116             {
117                 throw new InvalidOperationException("Frequencies validation failed.");
118             }
119             //else
120             //{//
121             //    if (value.Frequency > 0)
122             //    {
123             //        var frequency = value.Frequency;
124             //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
125             //        var count = _countLinkFrequency(linkIndex);
126             //        if ((frequency > count && frequency - count > 1) || (count > frequency
127             //            ↪ && count - frequency > 1))
128             //            throw new InvalidOperationException("Frequencies validation
129             //            ↪ failed.");
130             //    }
131             //}
132         }
133     }
134 }

```

#### 1.64 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Numbers;
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 LinkFrequency<TLink>
9     {
10         public TLink Frequency { get; set; }

```

```

11     public TLink Link { get; set; }
12
13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     public LinkFrequency(TLink frequency, TLink link)
15     {
16         Frequency = frequency;
17         Link = link;
18     }
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     public LinkFrequency() { }
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public override string ToString() => $"F: {Frequency}, L: {Link}";
31 }
32 }

```

## 1.65 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.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.Sequences.Frequencies.Cache
7 {
8     public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
9         ↳ IConverter<Doublet<TLink>, TLink>
10     {
11         private readonly LinkFrequenciesCache<TLink> _cache;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public
15             ↳ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
16             ↳ cache) => _cache = cache;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
20     }
21 }

```

## 1.66 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs

```

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> :
9         ↳ SequenceSymbolFrequencyOneOffCounter<TLink>
10     {
11         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
15             ↳ ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
16             : base(links, sequenceLink, symbol)
17             => _markedSequenceMatcher = markedSequenceMatcher;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public override TLink Count()
21         {
22             if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
23             {
24                 return default;
25             }
26             return base.Count();
27         }
28     }
29 }

```



## 1.67 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter

```

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 =
14             ↪ EqualityComparer<TLink>.Default;
15         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
16
17         protected readonly ILinks<TLink> _links;
18         protected readonly TLink _sequenceLink;
19         protected readonly TLink _symbol;
20         protected TLink _total;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
24             ↪ TLink symbol)
25         {
26             _links = links;
27             _sequenceLink = sequenceLink;
28             _symbol = symbol;
29             _total = default;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public virtual TLink Count()
34         {
35             if (_comparer.Compare(_total, default) > 0)
36             {
37                 return _total;
38             }
39             StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
40                 ↪ IsElement, VisitElement);
41             return _total;
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
46             ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
47             ↪ IsPartialPoint
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         private bool VisitElement(TLink element)
51         {
52             if (_equalityComparer.Equals(element, _symbol))
53             {
54                 _total = Arithmetic.Increment(_total);
55             }
56             return true;
57         }
58     }
59 }

```

## 1.68 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter

```

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,
15             ↪ ICriterionMatcher<TLink> markedSequenceMatcher)
16         {
17             _links = links;
18             _markedSequenceMatcher = markedSequenceMatcher;
19         }
20     }
21 }

```

```

19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     public TLink Count(TLink argument) => new
21     ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
22     ↪ _markedSequenceMatcher, argument).Count();
23 }

```

## 1.69 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter

```

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> :
10     ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
11     {
12         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
16         ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
17         : base(links, symbol)
18         => _markedSequenceMatcher = markedSequenceMatcher;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override void CountSequenceSymbolFrequency(TLink link)
22         {
23             var symbolFrequencyCounter = new
24             ↪ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
25             ↪ _markedSequenceMatcher, link, _symbol);
26             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
27         }
28     }
29 }

```

## 1.70 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter

```

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
17         ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
18     }
19 }

```

## 1.71 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter

```

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 =
13         ↪ EqualityComparer<TLink>.Default;
14         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
15
16         protected readonly ILinks<TLink> _links;
17         protected readonly TLink _symbol;
18         protected readonly HashSet<TLink> _visits;
19     }
20 }

```

```

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,
58             ↪ link, _symbol);
59         _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
60     }
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     private TLink EachElementHandler(IList<TLink> doublet)
64     {
65         var constants = _links.Constants;
66         var doubletIndex = doublet[constants.IndexPart];
67         if (_visits.Add(doubletIndex))
68         {
69             CountCore(doubletIndex);
70         }
71         return constants.Continue;
72     }
73 }

```

## 1.72 ./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 =
13             ↪ EqualityComparer<TLink>.Default;
14
15         private readonly TLink _heightPropertyMarker;
16         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
17         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
18         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
19         private readonly IProperties<TLink, TLink, TLink> _propertyOperator;
20
21         [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 }

```

### 1.73 ./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>,
10         ↪ ISequenceHeightProvider<TLink>
11     {
12         private readonly ICriterionMatcher<TLink> _elementMatcher;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
16             ↪ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public TLink Get(TLink sequence)
20         {
21             var height = default(TLink);
22             var pairOrElement = sequence;
23             while (!_elementMatcher.IsMatched(pairOrElement))
24             {
25                 pairOrElement = _links.GetTarget(pairOrElement);
26                 height = Arithmetic.Increment(height);
27             }
28             return height;
29         }
30     }
31 }

```

### 1.74 ./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 }

```

## 1.75 ./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 =
12             ↳ EqualityComparer<TLink>.Default;
13
14         private readonly LinkFrequenciesCache<TLink> _cache;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
18             ↳ _cache = cache;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public bool Add(IList<TLink> sequence)
22         {
23             var indexed = true;
24             var i = sequence.Count;
25             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
26                 ↳ { }
27             for (; i >= 1; i--)
28             {
29                 _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
30             }
31             return indexed;
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         private bool IsIndexedWithIncrement(TLink source, TLink target)
36         {
37             var frequency = _cache.GetFrequency(source, target);
38             if (frequency == null)
39             {
40                 return false;
41             }
42             var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
43             if (indexed)
44             {
45                 _cache.IncrementFrequency(source, target);
46             }
47             return indexed;
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public bool MightContain(IList<TLink> sequence)
52         {
53             var indexed = true;
54             var i = sequence.Count;
55             while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
56             return indexed;
57         }
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         private bool IsIndexed(TLink source, TLink target)
61         {
62             var frequency = _cache.GetFrequency(source, target);
63             if (frequency == null)
64             {
65                 return false;
66             }
67             return !_equalityComparer.Equals(frequency.Frequency, default);
68         }
69     }
70 }

```

## 1.76 ./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs

```

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

```

```

8 namespace Platform.Data.Doublets.Sequences.Indexes
9 {
10     public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
        ↳ ISequenceIndex<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
13
14         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
15         private readonly IIncrementer<TLink> _frequencyIncrementer;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IProperty<TLink, TLink>
            ↳ frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
            : base(links)
19         {
20             _frequencyPropertyOperator = frequencyPropertyOperator;
21             _frequencyIncrementer = frequencyIncrementer;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public override bool Add(IList<TLink> sequence)
26         {
27             var indexed = true;
28             var i = sequence.Count;
29             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
30                 ↳ { }
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.SearchOrCreate(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 }

```

## 1.77 ./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 }

```

## 1.78 ./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 =
11             ↪ EqualityComparer<TLink>.Default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public SequenceIndex(ILinks<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public virtual bool Add(IList<TLink> sequence)
18         {
19             var indexed = true;
20             var i = sequence.Count;
21             while (--i >= 1 && (indexed =
22                 ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
23                 ↪ default))) { }
24             for (; i >= 1; i--)
25             {
26                 _links.GetOrCreate(sequence[i - 1], sequence[i]);
27             }
28             return indexed;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public virtual bool MightContain(IList<TLink> sequence)
33         {
34             var indexed = true;
35             var i = sequence.Count;
36             while (--i >= 1 && (indexed =
37                 ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
38                 ↪ default))) { }
39             return indexed;
40         }
41     }
42 }

```

## 1.79 ./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 =
11             ↪ EqualityComparer<TLink>.Default;
12
13         private readonly ISynchronizedLinks<TLink> _links;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public bool Add(IList<TLink> sequence)
20         {
21             var indexed = true;
22             var i = sequence.Count;
23             var links = _links.Unsync;
24             _links.SyncRoot.ExecuteReadOperation(() =>
25             {
26                 while (--i >= 1 && (indexed =
27                     ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
28                     ↪ sequence[i]), default))) { }
29             });
30             if (!indexed)
31             {
32                 _links.SyncRoot.ExecuteWriteOperation(() =>
33                 {
34                     for (; i >= 1; i--)
35                     {

```

```

33         links.GetOrCreate(sequence[i - 1], sequence[i]);
34     }
35     });
36 }
37     return indexed;
38 }
39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 public bool MightContain(ICollection<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 =
49             ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
50             ↳ sequence[i]), default))) { }
51         return indexed;
52     });
53 }

```

#### 1.80 ./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.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 Unindex<TLink> : ISequenceIndex<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public virtual bool Add(ICollection<TLink> sequence) => false;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public virtual bool MightContain(ICollection<TLink> sequence) => true;
15     }
16 }

```

#### 1.81 ./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs

```

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, sequence.Length - 1);
43     });
44 }
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
48 {
49     #if DEBUG
50         if ((stopAt - startAt) < 0)
51         {
52             throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
53                 ↳ меньше или равен stopAt");
54         }
55         #endif
56         if ((stopAt - startAt) == 0)
57         {
58             return new[] { sequence[startAt] };
59         }
60         if ((stopAt - startAt) == 1)
61         {
62             return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
63         }
64         var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
65         var last = 0;
66         for (var splitter = startAt; splitter < stopAt; splitter++)
67         {
68             var left = CreateAllVariants2Core(sequence, startAt, splitter);
69             var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
70             for (var i = 0; i < left.Length; i++)
71             {
72                 for (var j = 0; j < right.Length; j++)
73                 {
74                     var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
75                     if (variant == Constants.Null)
76                     {
77                         throw new NotImplementedException("Creation cancellation is not
78                             ↳ implemented.");
79                     }
80                     variants[last++] = variant;
81                 }
82             }
83         }
84         return variants;
85     }
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 public List<ulong> CreateAllVariants1(params ulong[] sequence)
89 {
90     return _sync.ExecuteWriteOperation(() =>
91     {
92         if (sequence.IsNullOrEmpty())
93         {
94             return new List<ulong>();
95         }
96         Links.Unsync.EnsureLinkExists(sequence);
97         if (sequence.Length == 1)
98         {
99             return new List<ulong> { sequence[0] };
100         }
101         var results = new
102             ↳ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
103         return CreateAllVariants1Core(sequence, results);
104     });
105 }
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
109 {
110     if (sequence.Length == 2)
111     {
112         var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
113         if (link == Constants.Null)
114         {

```

```

112         throw new NotImplementedException("Creation cancellation is not
113         ↪ implemented.");
114     }
115     results.Add(link);
116     return results;
117 }
118 var innerSequenceLength = sequence.Length - 1;
119 var innerSequence = new ulong[innerSequenceLength];
120 for (var li = 0; li < innerSequenceLength; li++)
121 {
122     var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
123     if (link == Constants.Null)
124     {
125         throw new NotImplementedException("Creation cancellation is not
126         ↪ implemented.");
127     }
128     for (var isi = 0; isi < li; isi++)
129     {
130         innerSequence[isi] = sequence[isi];
131     }
132     innerSequence[li] = link;
133     for (var isi = li + 1; isi < innerSequenceLength; isi++)
134     {
135         innerSequence[isi] = sequence[isi + 1];
136     }
137     CreateAllVariants1Core(innerSequence, results);
138 }
139 return results;
140 }
141 #endregion
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 public HashSet<ulong> Each1(params ulong[] sequence)
144 {
145     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
146     Each1(link =>
147     {
148         if (!visitedLinks.Contains(link))
149         {
150             visitedLinks.Add(link); // изучить почему случаются повторы
151         }
152         return true;
153     }, sequence);
154     return visitedLinks;
155 }
156 [MethodImpl(MethodImplOptions.AggressiveInlining)]
157 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
158 {
159     if (sequence.Length == 2)
160     {
161         Links.Unsync.Each(sequence[0], sequence[1], handler);
162     }
163     else
164     {
165         var innerSequenceLength = sequence.Length - 1;
166         for (var li = 0; li < innerSequenceLength; li++)
167         {
168             var left = sequence[li];
169             var right = sequence[li + 1];
170             if (left == 0 && right == 0)
171             {
172                 continue;
173             }
174             var linkIndex = li;
175             ulong[] innerSequence = null;
176             Links.Unsync.Each(doublet =>
177             {
178                 if (innerSequence == null)
179                 {
180                     innerSequence = new ulong[innerSequenceLength];
181                     for (var isi = 0; isi < linkIndex; isi++)
182                     {
183                         innerSequence[isi] = sequence[isi];
184                     }
185                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
186                     {

```

```

188         innerSequence[isi] = sequence[isi + 1];
189     }
190 }
191     innerSequence[linkIndex] = doublet[Constants.IndexPart];
192     Each1(handler, innerSequence);
193     return Constants.Continue;
194 }, Constants.Any, left, right);
195 }
196 }
197 }
198
199 [MethodImpl(MethodImplOptions.AggressiveInlining)]
200 public HashSet<ulong> EachPart(params ulong[] sequence)
201 {
202     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
203     EachPartCore(link =>
204     {
205         var linkIndex = link[Constants.IndexPart];
206         if (!visitedLinks.Contains(linkIndex))
207         {
208             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
209         }
210         return Constants.Continue;
211     }, sequence);
212     return visitedLinks;
213 }
214
215 [MethodImpl(MethodImplOptions.AggressiveInlining)]
216 public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
217 {
218     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
219     EachPartCore(link =>
220     {
221         var linkIndex = link[Constants.IndexPart];
222         if (!visitedLinks.Contains(linkIndex))
223         {
224             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
225             return handler(new LinkAddress<LinkIndex>(linkIndex));
226         }
227         return Constants.Continue;
228     }, sequence);
229 }
230
231 [MethodImpl(MethodImplOptions.AggressiveInlining)]
232 private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
233 ↪ sequence)
234 {
235     if (sequence.IsNullOrEmpty())
236     {
237         return;
238     }
239     Links.EnsureLinkIsAnyOrExists(sequence);
240     if (sequence.Length == 1)
241     {
242         var link = sequence[0];
243         if (link > 0)
244         {
245             handler(new LinkAddress<LinkIndex>(link));
246         }
247         else
248         {
249             Links.Each(Constants.Any, Constants.Any, handler);
250         }
251     }
252     else if (sequence.Length == 2)
253     {
254         //_links.Each(sequence[0], sequence[1], handler);
255         //  o_|      x_o ...
256         // x_|      |__|
257         Links.Each(sequence[1], Constants.Any, doublet =>
258         {
259             var match = Links.SearchOrDefault(sequence[0], doublet);
260             if (match != Constants.Null)
261             {
262                 handler(new LinkAddress<LinkIndex>(match));
263             }
264             return true;
265         });

```

```

265 // |_x      ... x_o
266 // |_o      |__|
267 Links.Unsync.Each(Constants.Any, sequence[0], doublet =>
268 {
269     var match = Links.SearchOrDefault(doublet, sequence[1]);
270     if (match != 0)
271     {
272         handler(new LinkAddress<LinkIndex>(match));
273     }
274     return true;
275 });
276 //      .x o_.
277 //      |__|
278 PartialStepRight(x => handler(x), sequence[0], sequence[1]);
279 }
280 else
281 {
282     throw new NotImplementedException();
283 }
284 }
285
286 [MethodImpl(MethodImplOptions.AggressiveInlining)]
287 private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
288 {
289     Links.Unsync.Each(Constants.Any, left, doublet =>
290     {
291         StepRight(handler, doublet, right);
292         if (left != doublet)
293         {
294             PartialStepRight(handler, doublet, right);
295         }
296         return true;
297     });
298 }
299
300 [MethodImpl(MethodImplOptions.AggressiveInlining)]
301 private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
302 {
303     Links.Unsync.Each(left, Constants.Any, rightStep =>
304     {
305         TryStepRightUp(handler, right, rightStep);
306         return true;
307     });
308 }
309
310 [MethodImpl(MethodImplOptions.AggressiveInlining)]
311 private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
↪ stepFrom)
312 {
313     var upStep = stepFrom;
314     var firstSource = Links.Unsync.GetTarget(upStep);
315     while (firstSource != right && firstSource != upStep)
316     {
317         upStep = firstSource;
318         firstSource = Links.Unsync.GetSource(upStep);
319     }
320     if (firstSource == right)
321     {
322         handler(new LinkAddress<LinkIndex>(stepFrom));
323     }
324 }
325
326 // TODO: Test
327 [MethodImpl(MethodImplOptions.AggressiveInlining)]
328 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
329 {
330     Links.Unsync.Each(right, Constants.Any, doublet =>
331     {
332         StepLeft(handler, left, doublet);
333         if (right != doublet)
334         {
335             PartialStepLeft(handler, left, doublet);
336         }
337         return true;
338     });
339 }
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)

```

```

343 {
344     Links.Unsync.Each(Constants.Any, right, leftStep =>
345     {
346         TryStepLeftUp(handler, left, leftStep);
347         return true;
348     });
349 }
350
351 [MethodImpl(MethodImplOptions.AggressiveInlining)]
352 private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
353 {
354     var upStep = stepFrom;
355     var firstTarget = Links.Unsync.GetSource(upStep);
356     while (firstTarget != left && firstTarget != upStep)
357     {
358         upStep = firstTarget;
359         firstTarget = Links.Unsync.GetTarget(upStep);
360     }
361     if (firstTarget == left)
362     {
363         handler(new LinkAddress<LinkIndex>(stepFrom));
364     }
365 }
366
367 [MethodImpl(MethodImplOptions.AggressiveInlining)]
368 private bool StartsWith(ulong sequence, ulong link)
369 {
370     var upStep = sequence;
371     var firstSource = Links.Unsync.GetSource(upStep);
372     while (firstSource != link && firstSource != upStep)
373     {
374         upStep = firstSource;
375         firstSource = Links.Unsync.GetSource(upStep);
376     }
377     return firstSource == link;
378 }
379
380 [MethodImpl(MethodImplOptions.AggressiveInlining)]
381 private bool EndsWith(ulong sequence, ulong link)
382 {
383     var upStep = sequence;
384     var lastTarget = Links.Unsync.GetTarget(upStep);
385     while (lastTarget != link && lastTarget != upStep)
386     {
387         upStep = lastTarget;
388         lastTarget = Links.Unsync.GetTarget(upStep);
389     }
390     return lastTarget == link;
391 }
392
393 [MethodImpl(MethodImplOptions.AggressiveInlining)]
394 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
395 {
396     return _sync.ExecuteReadOperation(() =>
397     {
398         var results = new List<ulong>();
399         if (sequence.Length > 0)
400         {
401             Links.EnsureLinkExists(sequence);
402             var firstElement = sequence[0];
403             if (sequence.Length == 1)
404             {
405                 results.Add(firstElement);
406                 return results;
407             }
408             if (sequence.Length == 2)
409             {
410                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
411                 if (doublet != Constants.Null)
412                 {
413                     results.Add(doublet);
414                 }
415                 return results;
416             }
417             var linksInSequence = new HashSet<ulong>(sequence);
418             void handler(IList<LinkIndex> result)
419             {
420                 var resultIndex = result[Links.Constants.IndexPart];
421                 var filterPosition = 0;

```

```

422         StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
423         ↪ Links.Unsync.GetTarget,
424         ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
425         ↪ x =>
426         {
427             if (filterPosition == sequence.Length)
428             {
429                 filterPosition = -2; // Длиннее чем нужно
430                 return false;
431             }
432             if (x != sequence[filterPosition])
433             {
434                 filterPosition = -1;
435                 return false; // Начинается иначе
436             }
437             filterPosition++;
438             return true;
439         });
440     if (filterPosition == sequence.Length)
441     {
442         results.Add(resultIndex);
443     }
444     if (sequence.Length >= 2)
445     {
446         StepRight(handler, sequence[0], sequence[1]);
447     }
448     var last = sequence.Length - 2;
449     for (var i = 1; i < last; i++)
450     {
451         PartialStepRight(handler, sequence[i], sequence[i + 1]);
452     }
453     if (sequence.Length >= 3)
454     {
455         StepLeft(handler, sequence[sequence.Length - 2],
456         ↪ sequence[sequence.Length - 1]);
457     }
458     }
459     return results;
460 }
461
462 [MethodImpl(MethodImplOptions.AggressiveInlining)]
463 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
464 {
465     return _sync.ExecuteReadOperation(() =>
466     {
467         var results = new HashSet<ulong>();
468         if (sequence.Length > 0)
469         {
470             Links.EnsureLinkExists(sequence);
471             var firstElement = sequence[0];
472             if (sequence.Length == 1)
473             {
474                 results.Add(firstElement);
475                 return results;
476             }
477             if (sequence.Length == 2)
478             {
479                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
480                 if (doublet != Constants.Null)
481                 {
482                     results.Add(doublet);
483                 }
484                 return results;
485             }
486             var matcher = new Matcher(this, sequence, results, null);
487             if (sequence.Length >= 2)
488             {
489                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
490             }
491             var last = sequence.Length - 2;
492             for (var i = 1; i < last; i++)
493             {
494                 PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
495                 ↪ sequence[i + 1]);

```

```

496         if (sequence.Length >= 3)
497         {
498             StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
499                 ↪ sequence[sequence.Length - 1]);
500         }
501         return results;
502     });
503 }
504
505 public const int MaxSequenceFormatSize = 200;
506
507 [MethodImpl(MethodImplOptions.AggressiveInlining)]
508 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
509     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
510
511 [MethodImpl(MethodImplOptions.AggressiveInlining)]
512 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
513     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
514     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
515     ↪ elementToString, insertComma, knownElements));
516
517 [MethodImpl(MethodImplOptions.AggressiveInlining)]
518 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
519     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
520     ↪ LinkIndex[] knownElements)
521 {
522     var linksInSequence = new HashSet<ulong>(knownElements);
523     //var entered = new HashSet<ulong>();
524     var sb = new StringBuilder();
525     sb.Append('{');
526     if (links.Exists(sequenceLink))
527     {
528         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
529             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
530             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
531         {
532             if (insertComma && sb.Length > 1)
533             {
534                 sb.Append(',');
535             }
536             //if (entered.Contains(element))
537             //{
538             //    sb.Append('{');
539             //    elementToString(sb, element);
540             //    sb.Append('}');
541             //}
542             //else
543             elementToString(sb, element);
544             if (sb.Length < MaxSequenceFormatSize)
545             {
546                 return true;
547             }
548             sb.Append(insertComma ? ", ..." : "...");
549             return false;
550         });
551     }
552     sb.Append('}');
553     return sb.ToString();
554 }
555
556 [MethodImpl(MethodImplOptions.AggressiveInlining)]
557 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
558     ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
559     ↪ knownElements);
560
561 [MethodImpl(MethodImplOptions.AggressiveInlining)]
562 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
563     ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
564     ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
565     ↪ sequenceLink, elementToString, insertComma, knownElements));
566
567 [MethodImpl(MethodImplOptions.AggressiveInlining)]
568 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
569     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
570     ↪ LinkIndex[] knownElements)
571 {
572     var linksInSequence = new HashSet<ulong>(knownElements);

```

```

559     var entered = new HashSet<ulong>();
560     var sb = new StringBuilder();
561     sb.Append('{');
562     if (links.Exists(sequenceLink))
563     {
564         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
565             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
566             entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
567             {
568                 if (insertComma && sb.Length > 1)
569                 {
570                     sb.Append(',');
571                 }
572                 if (entered.Contains(element))
573                 {
574                     sb.Append('{');
575                     elementToString(sb, element);
576                     sb.Append('}');
577                 }
578                 else
579                 {
580                     elementToString(sb, element);
581                 }
582                 if (sb.Length < MaxSequenceFormatSize)
583                 {
584                     return true;
585                 }
586                 sb.Append(insertComma ? ", ..." : "...");
587                 return false;
588             });
589     }
590     sb.Append('}');
591     return sb.ToString();
592 }
593 [MethodImpl(MethodImplOptions.AggressiveInlining)]
594 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
595 {
596     return _sync.ExecuteReadOperation(() =>
597     {
598         if (sequence.Length > 0)
599         {
600             Links.EnsureLinkExists(sequence);
601             var results = new HashSet<ulong>();
602             for (var i = 0; i < sequence.Length; i++)
603             {
604                 AllUsagesCore(sequence[i], results);
605             }
606             var filteredResults = new List<ulong>();
607             var linksInSequence = new HashSet<ulong>(sequence);
608             foreach (var result in results)
609             {
610                 var filterPosition = -1;
611                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
612                     Links.Unsync.GetTarget,
613                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
614                     x =>
615                     {
616                         if (filterPosition == (sequence.Length - 1))
617                         {
618                             return false;
619                         }
620                         if (filterPosition >= 0)
621                         {
622                             if (x == sequence[filterPosition + 1])
623                             {
624                                 filterPosition++;
625                             }
626                             else
627                             {
628                                 return false;
629                             }
630                         }
631                         if (filterPosition < 0)
632                         {
633                             if (x == sequence[0])
634                             {
635                                 filterPosition = 0;
636                             }
637                         }
638                     }
639             }
640             filteredResults.AddRange(filteredResults);
641         }
642     });
643 }

```



```

634         }
635     }
636     return true;
637 });
638 if (filterPosition == (sequence.Length - 1))
639 {
640     filteredResults.Add(result);
641 }
642 }
643 return filteredResults;
644 }
645 return new List<ulong>();
646 });
647 }
648
649 [MethodImpl(MethodImplOptions.AggressiveInlining)]
650 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
651 {
652     return _sync.ExecuteReadOperation(() =>
653     {
654         if (sequence.Length > 0)
655         {
656             Links.EnsureLinkExists(sequence);
657             var results = new HashSet<ulong>();
658             for (var i = 0; i < sequence.Length; i++)
659             {
660                 AllUsagesCore(sequence[i], results);
661             }
662             var filteredResults = new HashSet<ulong>();
663             var matcher = new Matcher(this, sequence, filteredResults, null);
664             matcher.AddAllPartialMatchedToResults(results);
665             return filteredResults;
666         }
667         return new HashSet<ulong>();
668     });
669 }
670
671 [MethodImpl(MethodImplOptions.AggressiveInlining)]
672 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
673 ↪ params ulong[] sequence)
674 {
675     return _sync.ExecuteReadOperation(() =>
676     {
677         if (sequence.Length > 0)
678         {
679             Links.EnsureLinkExists(sequence);
680
681             var results = new HashSet<ulong>();
682             var filteredResults = new HashSet<ulong>();
683             var matcher = new Matcher(this, sequence, filteredResults, handler);
684             for (var i = 0; i < sequence.Length; i++)
685             {
686                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
687                 {
688                     return false;
689                 }
690             }
691             return true;
692         }
693         return true;
694     });
695 }
696
697 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
698 //{
699 //    return Sync.ExecuteReadOperation(() =>
700 //    {
701 //        if (sequence.Length > 0)
702 //        {
703 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
704 //
705 //            var firstResults = new HashSet<ulong>();
706 //            var lastResults = new HashSet<ulong>();
707 //
708 //            var first = sequence.First(x => x != LinksConstants.Any);
709 //            var last = sequence.Last(x => x != LinksConstants.Any);
710 //
711 //            AllUsagesCore(first, firstResults);
712 //            AllUsagesCore(last, lastResults);

```

```

712 //                firstResults.IntersectWith(lastResults);
713 //
714 //                //for (var i = 0; i < sequence.Length; i++)
715 //                //    AllUsagesCore(sequence[i], results);
716 //
717 //                var filteredResults = new HashSet<ulong>();
718 //                var matcher = new Matcher(this, sequence, filteredResults, null);
719 //                matcher.AddAllPartialMatchedToResults(firstResults);
720 //                return filteredResults;
721 //            }
722 //
723 //        return new HashSet<ulong>();
724 //    });
725 //}
726
727 [MethodImpl(MethodImplOptions.AggressiveInlining)]
728 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
729 {
730     return _sync.ExecuteReadOperation((Func<HashSet<ulong>>)(() =>
731     {
732         if (sequence.Length > 0)
733         {
734             ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
735                 ↪ (IList<ulong>)sequence);
736             var firstResults = new HashSet<ulong>();
737             var lastResults = new HashSet<ulong>();
738             var first = sequence.First(x => x != Constants.Any);
739             var last = sequence.Last(x => x != Constants.Any);
740             AllUsagesCore(first, firstResults);
741             AllUsagesCore(last, lastResults);
742             firstResults.IntersectWith(lastResults);
743             //for (var i = 0; i < sequence.Length; i++)
744             //    AllUsagesCore(sequence[i], results);
745             var filteredResults = new HashSet<ulong>();
746             var matcher = new Matcher(this, sequence, filteredResults, null);
747             matcher.AddAllPartialMatchedToResults(firstResults);
748             return filteredResults;
749         }
750         return new HashSet<ulong>();
751     }));
752 }
753
754 [MethodImpl(MethodImplOptions.AggressiveInlining)]
755 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
756     ↪ IList<ulong> sequence)
757 {
758     return _sync.ExecuteReadOperation(() =>
759     {
760         if (sequence.Count > 0)
761         {
762             Links.EnsureLinkExists(sequence);
763             var results = new HashSet<LinkIndex>();
764             //var nextResults = new HashSet<ulong>();
765             //for (var i = 0; i < sequence.Length; i++)
766             //{
767             //    AllUsagesCore(sequence[i], nextResults);
768             //    if (results.IsNullOrEmpty())
769             //    {
770             //        results = nextResults;
771             //        nextResults = new HashSet<ulong>();
772             //    }
773             //    else
774             //    {
775             //        results.IntersectWith(nextResults);
776             //        nextResults.Clear();
777             //    }
778             //}
779             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
780             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
781             var next = new HashSet<ulong>();
782             for (var i = 1; i < sequence.Count; i++)
783             {
784                 var collector = new AllUsagesCollector1(Links.Unsync, next);
785                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
786
787                 results.IntersectWith(next);
788                 next.Clear();

```

```

788     }
789     var filteredResults = new HashSet<ulong>();
790     var matcher = new Matcher(this, sequence, filteredResults, null,
791         ↪ readAsElements);
792     matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
793         ↪ x)); // OrderBy is a Hack
794     return filteredResults;
795 }
796 }
797
798 // Does not work
799 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
800     ↪ params ulong[] sequence)
801 //{
802 //    var visited = new HashSet<ulong>();
803 //    var results = new HashSet<ulong>();
804 //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
805     ↪ true; }, readAsElements);
806 //    var last = sequence.Length - 1;
807 //    for (var i = 0; i < last; i++)
808 //    {
809 //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
810 //    }
811 //    return results;
812 //}
813
814 [MethodImpl(MethodImplOptions.AggressiveInlining)]
815 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
816 {
817     return _sync.ExecuteReadOperation(() =>
818     {
819         if (sequence.Length > 0)
820         {
821             Links.EnsureLinkExists(sequence);
822             //var firstElement = sequence[0];
823             //if (sequence.Length == 1)
824             //{
825             //    //results.Add(firstElement);
826             //    return results;
827             //}
828             //if (sequence.Length == 2)
829             //{
830             //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
831             //    //if (doublet != Doublets.Links.Null)
832             //    //    results.Add(doublet);
833             //    return results;
834             //}
835             //var lastElement = sequence[sequence.Length - 1];
836             //Func<ulong, bool> handler = x =>
837             //{
838             //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
839             //        ↪ results.Add(x);
840             //    return true;
841             //};
842             //if (sequence.Length >= 2)
843             //    StepRight(handler, sequence[0], sequence[1]);
844             //var last = sequence.Length - 2;
845             //for (var i = 1; i < last; i++)
846             //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
847             //if (sequence.Length >= 3)
848             //    StepLeft(handler, sequence[sequence.Length - 2],
849             //        ↪ sequence[sequence.Length - 1]);
850             //if (sequence.Length == 1)
851             //{
852             //    throw new NotImplementedException(); // all sequences, containing
853             //        ↪ this element?
854             //}
855             //if (sequence.Length == 2)
856             //{
857             //    var results = new List<ulong>();
858             //    PartialStepRight(results.Add, sequence[0], sequence[1]);
859             //    return results;
860             //}
861             //var matches = new List<List<ulong>>();
862             //var last = sequence.Length - 1;

```

```

858         for (var i = 0; i < last; i++)
859         {
860             var results = new List<ulong>();
861             //StepRight(results.Add, sequence[i], sequence[i + 1]);
862             PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
863             if (results.Count > 0)
864                 matches.Add(results);
865             else
866                 return results;
867             if (matches.Count == 2)
868             {
869                 var merged = new List<ulong>();
870                 for (var j = 0; j < matches[0].Count; j++)
871                     for (var k = 0; k < matches[1].Count; k++)
872                         CloseInnerConnections(merged.Add, matches[0][j],
873 → matches[1][k]);
874                 if (merged.Count > 0)
875                     matches = new List<List<ulong>> { merged };
876                 else
877                     return new List<ulong>();
878             }
879             if (matches.Count > 0)
880             {
881                 var usages = new HashSet<ulong>();
882                 for (int i = 0; i < sequence.Length; i++)
883                 {
884                     AllUsagesCore(sequence[i], usages);
885                 }
886                 //for (int i = 0; i < matches[0].Count; i++)
887                 //    AllUsagesCore(matches[0][i], usages);
888                 //usages.UnionWith(matches[0]);
889                 return usages.ToList();
890             }
891             var firstLinkUsages = new HashSet<ulong>();
892             AllUsagesCore(sequence[0], firstLinkUsages);
893             firstLinkUsages.Add(sequence[0]);
894             //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
895             //    sequence[0] }; // or all sequences, containing this element?
896             //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
897             //    sequence[0], 1).ToList();
898             var results = new HashSet<ulong>();
899             foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
900             //    firstLinkUsages, 1))
901             {
902                 AllUsagesCore(match, results);
903             }
904             return results.ToList();
905         }
906     }
907     return new List<ulong>();
908 }
909
910 /// <remarks>
911 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
912 /// </remarks>
913 [MethodImpl(MethodImplOptions.AggressiveInlining)]
914 public HashSet<ulong> AllUsages(ulong link)
915 {
916     return _sync.ExecuteReadOperation(() =>
917     {
918         var usages = new HashSet<ulong>();
919         AllUsagesCore(link, usages);
920         return usages;
921     });
922 }
923
924 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
925 // той связи с которой начинался поиск (STTTSSSTT),
926 // причём достаточно одного бита для хранения перехода влево или вправо
927 [MethodImpl(MethodImplOptions.AggressiveInlining)]
928 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
929 {
930     bool handler(ulong doublet)
931     {
932         if (usages.Add(doublet))
933         {
934             // ...
935         }
936     }
937 }

```

```

930         AllUsagesCore(doublet, usages);
931     }
932     return true;
933 }
934 Links.Unsync.Each(link, Constants.Any, handler);
935 Links.Unsync.Each(Constants.Any, link, handler);
936 }
937
938 [MethodImpl(MethodImplOptions.AggressiveInlining)]
939 public HashSet<ulong> AllBottomUsages(ulong link)
940 {
941     return _sync.ExecuteReadOperation(() =>
942     {
943         var visits = new HashSet<ulong>();
944         var usages = new HashSet<ulong>();
945         AllBottomUsagesCore(link, visits, usages);
946         return usages;
947     });
948 }
949
950 [MethodImpl(MethodImplOptions.AggressiveInlining)]
951 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
    ↳ usages)
952 {
953     bool handler(ulong doublet)
954     {
955         if (visits.Add(doublet))
956         {
957             AllBottomUsagesCore(doublet, visits, usages);
958         }
959         return true;
960     }
961     if (Links.Unsync.Count(Constants.Any, link) == 0)
962     {
963         usages.Add(link);
964     }
965     else
966     {
967         Links.Unsync.Each(link, Constants.Any, handler);
968         Links.Unsync.Each(Constants.Any, link, handler);
969     }
970 }
971
972 [MethodImpl(MethodImplOptions.AggressiveInlining)]
973 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
974 {
975     if (Options.UseSequenceMarker)
976     {
977         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
    ↳ Options.MarkedSequenceMatcher, symbol);
978         return counter.Count();
979     }
980     else
981     {
982         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
    ↳ symbol);
983         return counter.Count();
984     }
985 }
986
987 [MethodImpl(MethodImplOptions.AggressiveInlining)]
988 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
    ↳ LinkIndex> outerHandler)
989 {
990     bool handler(ulong doublet)
991     {
992         if (usages.Add(doublet))
993         {
994             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
995             {
996                 return false;
997             }
998             if (!AllUsagesCore1(doublet, usages, outerHandler))
999             {
1000                 return false;
1001             }
1002         }
1003         return true;

```

```

1004     }
1005     return Links.Unsync.Each(link, Constants.Any, handler)
1006         && Links.Unsync.Each(Constants.Any, link, handler);
1007 }
1008
1009 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1010 public void CalculateAllUsages(ulong[] totals)
1011 {
1012     var calculator = new AllUsagesCalculator(Links, totals);
1013     calculator.Calculate();
1014 }
1015
1016 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1017 public void CalculateAllUsages2(ulong[] totals)
1018 {
1019     var calculator = new AllUsagesCalculator2(Links, totals);
1020     calculator.Calculate();
1021 }
1022
1023 private class AllUsagesCalculator
1024 {
1025     private readonly SynchronizedLinks<ulong> _links;
1026     private readonly ulong[] _totals;
1027
1028     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1030     {
1031         _links = links;
1032         _totals = totals;
1033     }
1034
1035     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1036     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1037         ↪ CalculateCore);
1038
1039     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1040     private bool CalculateCore(ulong link)
1041     {
1042         if (_totals[link] == 0)
1043         {
1044             var total = 1UL;
1045             _totals[link] = total;
1046             var visitedChildren = new HashSet<ulong>();
1047             bool linkCalculator(ulong child)
1048             {
1049                 if (link != child && visitedChildren.Add(child))
1050                 {
1051                     total += _totals[child] == 0 ? 1 : _totals[child];
1052                 }
1053                 return true;
1054             }
1055             _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1056             _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1057             _totals[link] = total;
1058         }
1059         return true;
1060     }
1061 }
1062
1063 private class AllUsagesCalculator2
1064 {
1065     private readonly SynchronizedLinks<ulong> _links;
1066     private readonly ulong[] _totals;
1067
1068     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1069     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1070     {
1071         _links = links;
1072         _totals = totals;
1073     }
1074
1075     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1076     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1077         ↪ CalculateCore);
1078
1079     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1080     private bool IsElement(ulong link)
1081     {
1082         // _linksInSequence.Contains(link) ||

```

```

1081         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1082             ↪ link;
1083     }
1084     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1085     private bool CalculateCore(ulong link)
1086     {
1087         // TODO: Проработать защиту от заикливания
1088         // Основано на SequenceWalker.WalkLeft
1089         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1090         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1091         Func<ulong, bool> isElement = IsElement;
1092         void visitLeaf(ulong parent)
1093         {
1094             if (link != parent)
1095             {
1096                 _totals[parent]++;
1097             }
1098         }
1099         void visitNode(ulong parent)
1100         {
1101             if (link != parent)
1102             {
1103                 _totals[parent]++;
1104             }
1105         }
1106         var stack = new Stack();
1107         var element = link;
1108         if (isElement(element))
1109         {
1110             visitLeaf(element);
1111         }
1112         else
1113         {
1114             while (true)
1115             {
1116                 if (isElement(element))
1117                 {
1118                     if (stack.Count == 0)
1119                     {
1120                         break;
1121                     }
1122                     element = stack.Pop();
1123                     var source = getSource(element);
1124                     var target = getTarget(element);
1125                     // Обработка элемента
1126                     if (isElement(target))
1127                     {
1128                         visitLeaf(target);
1129                     }
1130                     if (isElement(source))
1131                     {
1132                         visitLeaf(source);
1133                     }
1134                     element = source;
1135                 }
1136                 else
1137                 {
1138                     stack.Push(element);
1139                     visitNode(element);
1140                     element = getTarget(element);
1141                 }
1142             }
1143             _totals[link]++;
1144             return true;
1145         }
1146     }
1147 }
1148
1149 private class AllUsagesCollector
1150 {
1151     private readonly ILinks<ulong> _links;
1152     private readonly HashSet<ulong> _usages;
1153
1154     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1155     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1156     {
1157         _links = links;
1158         _usages = usages;

```

```

1159     }
1160
1161     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1162     public bool Collect(ulong link)
1163     {
1164         if (_usages.Add(link))
1165         {
1166             _links.Each(link, _links.Constants.Any, Collect);
1167             _links.Each(_links.Constants.Any, link, Collect);
1168         }
1169         return true;
1170     }
1171 }
1172
1173 private class AllUsagesCollector1
1174 {
1175     private readonly ILinks<ulong> _links;
1176     private readonly HashSet<ulong> _usages;
1177     private readonly ulong _continue;
1178
1179     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1180     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1181     {
1182         _links = links;
1183         _usages = usages;
1184         _continue = _links.Constants.Continue;
1185     }
1186
1187     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1188     public ulong Collect(IList<ulong> link)
1189     {
1190         var linkIndex = _links.GetIndex(link);
1191         if (_usages.Add(linkIndex))
1192         {
1193             _links.Each(Collect, _links.Constants.Any, linkIndex);
1194         }
1195         return _continue;
1196     }
1197 }
1198
1199 private class AllUsagesCollector2
1200 {
1201     private readonly ILinks<ulong> _links;
1202     private readonly BitString _usages;
1203
1204     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1205     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1206     {
1207         _links = links;
1208         _usages = usages;
1209     }
1210
1211     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1212     public bool Collect(ulong link)
1213     {
1214         if (_usages.Add((long)link))
1215         {
1216             _links.Each(link, _links.Constants.Any, Collect);
1217             _links.Each(_links.Constants.Any, link, Collect);
1218         }
1219         return true;
1220     }
1221 }
1222
1223 private class AllUsagesIntersectingCollector
1224 {
1225     private readonly SynchronizedLinks<ulong> _links;
1226     private readonly HashSet<ulong> _intersectWith;
1227     private readonly HashSet<ulong> _usages;
1228     private readonly HashSet<ulong> _enter;
1229
1230     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1231     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
↪ intersectWith, HashSet<ulong> usages)
1232     {
1233         _links = links;
1234         _intersectWith = intersectWith;
1235         _usages = usages;
1236         _enter = new HashSet<ulong>(); // защита от заикливания
1237     }

```



```

1238 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1239 public bool Collect(ulong link)
1240 {
1241     if (_enter.Add(link))
1242     {
1243         if (_intersectWith.Contains(link))
1244         {
1245             _usages.Add(link);
1246         }
1247         _links.Unsync.Each(link, _links.Constants.Any, Collect);
1248         _links.Unsync.Each(_links.Constants.Any, link, Collect);
1249     }
1250     return true;
1251 }
1252 }
1253
1254 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1255 private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
1256     ↪ right)
1257 {
1258     TryStepLeftUp(handler, left, right);
1259     TryStepRightUp(handler, right, left);
1260 }
1261
1262 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1263 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
1264     ↪ right)
1265 {
1266     // Direct
1267     if (left == right)
1268     {
1269         handler(new LinkAddress<LinkIndex>(left));
1270     }
1271     var doublet = Links.Unsync.SearchOrDefault(left, right);
1272     if (doublet != Constants.Null)
1273     {
1274         handler(new LinkAddress<LinkIndex>(doublet));
1275     }
1276     // Inner
1277     CloseInnerConnections(handler, left, right);
1278     // Outer
1279     StepLeft(handler, left, right);
1280     StepRight(handler, left, right);
1281     PartialStepRight(handler, left, right);
1282     PartialStepLeft(handler, left, right);
1283 }
1284
1285 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1286 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
1287     ↪ HashSet<ulong> previousMatchings, long startAt)
1288 {
1289     if (startAt >= sequence.Length) // ?
1290     {
1291         return previousMatchings;
1292     }
1293     var secondLinkUsages = new HashSet<ulong>();
1294     AllUsagesCore(sequence[startAt], secondLinkUsages);
1295     secondLinkUsages.Add(sequence[startAt]);
1296     var matchings = new HashSet<ulong>();
1297     var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1298     //for (var i = 0; i < previousMatchings.Count; i++)
1299     foreach (var secondLinkUsage in secondLinkUsages)
1300     {
1301         foreach (var previousMatching in previousMatchings)
1302         {
1303             //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1304             ↪ secondLinkUsage);
1305             StepRight(filler.AddFirstAndReturnConstant, previousMatching,
1306             ↪ secondLinkUsage);
1307             TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
1308             ↪ previousMatching);
1309             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1310             ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1311             ↪ желаемым результатам.
1312             PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
1313             ↪ secondLinkUsage);

```

```

1306     }
1307 }
1308 if (matchings.Count == 0)
1309 {
1310     return matchings;
1311 }
1312 return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1313 }
1314
1315 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1316 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
↪ links, params ulong[] sequence)
1317 {
1318     if (sequence == null)
1319     {
1320         return;
1321     }
1322     for (var i = 0; i < sequence.Length; i++)
1323     {
1324         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
↪ !links.Exists(sequence[i]))
1325         {
1326             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
↪ $"patternSequence[{i}]");
1327         }
1328     }
1329 }
1330
1331 // Pattern Matching -> Key To Triggers
1332 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1333 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1334 {
1335     return _sync.ExecuteReadOperation(() =>
1336     {
1337         patternSequence = Simplify(patternSequence);
1338         if (patternSequence.Length > 0)
1339         {
1340             EnsureEachLinkIsAnyOrZeroOrManyOrExists(links, patternSequence);
1341             var uniqueSequenceElements = new HashSet<ulong>();
1342             for (var i = 0; i < patternSequence.Length; i++)
1343             {
1344                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
↪ ZeroOrMany)
1345                 {
1346                     uniqueSequenceElements.Add(patternSequence[i]);
1347                 }
1348             }
1349             var results = new HashSet<ulong>();
1350             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1351             {
1352                 AllUsagesCore(uniqueSequenceElement, results);
1353             }
1354             var filteredResults = new HashSet<ulong>();
1355             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1356             matcher.AddAllPatternMatchedToResults(results);
1357             return filteredResults;
1358         }
1359         return new HashSet<ulong>();
1360     });
1361 }
1362
1363 // Найти все возможные связи между указанным списком связей.
1364 // Находит связи между всеми указанными связями в любом порядке.
1365 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
↪ несколько раз в последовательности)
1366 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1367 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1368 {
1369     return _sync.ExecuteReadOperation(() =>
1370     {
1371         var results = new HashSet<ulong>();
1372         if (linksToConnect.Length > 0)
1373         {
1374             links.EnsureLinkExists(linksToConnect);
1375             AllUsagesCore(linksToConnect[0], results);
1376             for (var i = 1; i < linksToConnect.Length; i++)
1377             {
1378                 var next = new HashSet<ulong>();

```

```

1379         AllUsagesCore(linksToConnect[i], next);
1380         results.IntersectWith(next);
1381     }
1382 }
1383     return results;
1384 });
1385 }
1386
1387 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1388 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1389 {
1390     return _sync.ExecuteReadOperation(() =>
1391     {
1392         var results = new HashSet<ulong>();
1393         if (linksToConnect.Length > 0)
1394         {
1395             Links.EnsureLinkExists(linksToConnect);
1396             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1397             collector1.Collect(linksToConnect[0]);
1398             var next = new HashSet<ulong>();
1399             for (var i = 1; i < linksToConnect.Length; i++)
1400             {
1401                 var collector = new AllUsagesCollector(Links.Unsync, next);
1402                 collector.Collect(linksToConnect[i]);
1403                 results.IntersectWith(next);
1404                 next.Clear();
1405             }
1406             return results;
1407         }
1408     });
1409 }
1410
1411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1412 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1413 {
1414     return _sync.ExecuteReadOperation(() =>
1415     {
1416         var results = new HashSet<ulong>();
1417         if (linksToConnect.Length > 0)
1418         {
1419             Links.EnsureLinkExists(linksToConnect);
1420             var collector1 = new AllUsagesCollector(Links, results);
1421             collector1.Collect(linksToConnect[0]);
1422             //AllUsagesCore(linksToConnect[0], results);
1423             for (var i = 1; i < linksToConnect.Length; i++)
1424             {
1425                 var next = new HashSet<ulong>();
1426                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1427                 collector.Collect(linksToConnect[i]);
1428                 //AllUsagesCore(linksToConnect[i], next);
1429                 //results.IntersectWith(next);
1430                 results = next;
1431             }
1432             return results;
1433         }
1434     });
1435 }
1436
1437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1438 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1439 {
1440     return _sync.ExecuteReadOperation(() =>
1441     {
1442         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1443         ↪ BitArray((int)_links.Total + 1);
1444         if (linksToConnect.Length > 0)
1445         {
1446             Links.EnsureLinkExists(linksToConnect);
1447             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1448             collector1.Collect(linksToConnect[0]);
1449             for (var i = 1; i < linksToConnect.Length; i++)
1450             {
1451                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1452                 ↪ BitArray((int)_links.Total + 1);
1453                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1454                 collector.Collect(linksToConnect[i]);
1455                 results = results.And(next);
1456             }
1457         }
1458     });
1459 }

```

```

1455     }
1456     return results.GetSetUInt64Indices();
1457 });
1458 }
1459
1460 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1461 private static ulong[] Simplify(ulong[] sequence)
1462 {
1463     // Считаем новый размер последовательности
1464     long newLength = 0;
1465     var zeroOrManyStepped = false;
1466     for (var i = 0; i < sequence.Length; i++)
1467     {
1468         if (sequence[i] == ZeroOrMany)
1469         {
1470             if (zeroOrManyStepped)
1471             {
1472                 continue;
1473             }
1474             zeroOrManyStepped = true;
1475         }
1476         else
1477         {
1478             //if (zeroOrManyStepped) Is it efficient?
1479             zeroOrManyStepped = false;
1480         }
1481         newLength++;
1482     }
1483     // Строим новую последовательность
1484     zeroOrManyStepped = false;
1485     var newSequence = new ulong[newLength];
1486     long j = 0;
1487     for (var i = 0; i < sequence.Length; i++)
1488     {
1489         //var current = zeroOrManyStepped;
1490         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1491         //if (current && zeroOrManyStepped)
1492         //    continue;
1493         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1494         //if (zeroOrManyStepped && newZeroOrManyStepped)
1495         //    continue;
1496         //zeroOrManyStepped = newZeroOrManyStepped;
1497         if (sequence[i] == ZeroOrMany)
1498         {
1499             if (zeroOrManyStepped)
1500             {
1501                 continue;
1502             }
1503             zeroOrManyStepped = true;
1504         }
1505         else
1506         {
1507             //if (zeroOrManyStepped) Is it efficient?
1508             zeroOrManyStepped = false;
1509         }
1510         newSequence[j++] = sequence[i];
1511     }
1512     return newSequence;
1513 }
1514
1515 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1516 public static void TestSimplify()
1517 {
1518     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1519     ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1520     var simplifiedSequence = Simplify(sequence);
1521 }
1522
1523 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1524 public List<ulong> GetSimilarSequences() => new List<ulong>();
1525
1526 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1527 public void Prediction()
1528 {
1529     //_links
1530     //sequences
1531 }
1532
1533 #region From Triplets

```

```

1533 //public static void DeleteSequence(Link sequence)
1534 //{
1535 //}
1536
1537
1538 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1539 public List<ulong> CollectMatchingSequences(ulong[] links)
1540 {
1541     if (links.Length == 1)
1542     {
1543         throw new InvalidOperationException("Подпоследовательности с одним элементом не
1544             ↳ поддерживаются.");
1545     }
1546     var leftBound = 0;
1547     var rightBound = links.Length - 1;
1548     var left = links[leftBound++];
1549     var right = links[rightBound--];
1550     var results = new List<ulong>();
1551     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1552     return results;
1553 }
1554
1555 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1556 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1557     ↳ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1558 {
1559     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1560     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1561     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1562     {
1563         var nextLeftLink = middleLinks[leftBound];
1564         var elements = GetRightElements(leftLink, nextLeftLink);
1565         if (leftBound <= rightBound)
1566         {
1567             for (var i = elements.Length - 1; i >= 0; i--)
1568             {
1569                 var element = elements[i];
1570                 if (element != 0)
1571                 {
1572                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1573                         ↳ rightLink, rightBound, ref results);
1574                 }
1575             }
1576         }
1577         else
1578         {
1579             for (var i = elements.Length - 1; i >= 0; i--)
1580             {
1581                 var element = elements[i];
1582                 if (element != 0)
1583                 {
1584                     results.Add(element);
1585                 }
1586             }
1587         }
1588     }
1589     else
1590     {
1591         var nextRightLink = middleLinks[rightBound];
1592         var elements = GetLeftElements(rightLink, nextRightLink);
1593         if (leftBound <= rightBound)
1594         {
1595             for (var i = elements.Length - 1; i >= 0; i--)
1596             {
1597                 var element = elements[i];
1598                 if (element != 0)
1599                 {
1600                     CollectMatchingSequences(leftLink, leftBound, middleLinks,
1601                         ↳ elements[i], rightBound - 1, ref results);
1602                 }
1603             }
1604         }
1605         else
1606         {
1607             for (var i = elements.Length - 1; i >= 0; i--)
1608             {
1609                 var element = elements[i];
1610                 if (element != 0)

```

```

1607         {
1608             results.Add(element);
1609         }
1610     }
1611 }
1612
1613 }
1614
1615 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1616 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1617 {
1618     var result = new ulong[5];
1619     TryStepRight(startLink, rightLink, result, 0);
1620     Links.Each(Constants.Any, startLink, couple =>
1621     {
1622         if (couple != startLink)
1623         {
1624             if (TryStepRight(couple, rightLink, result, 2))
1625             {
1626                 return false;
1627             }
1628         }
1629         return true;
1630     });
1631     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1632     {
1633         result[4] = startLink;
1634     }
1635     return result;
1636 }
1637
1638 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1639 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1640 {
1641     var added = 0;
1642     Links.Each(startLink, Constants.Any, couple =>
1643     {
1644         if (couple != startLink)
1645         {
1646             var coupleTarget = Links.GetTarget(couple);
1647             if (coupleTarget == rightLink)
1648             {
1649                 result[offset] = couple;
1650                 if (++added == 2)
1651                 {
1652                     return false;
1653                 }
1654             }
1655             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1656                 ↪ == Net.And &&
1657             {
1658                 result[offset + 1] = couple;
1659                 if (++added == 2)
1660                 {
1661                     return false;
1662                 }
1663             }
1664         }
1665         return true;
1666     });
1667     return added > 0;
1668 }
1669
1670 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1671 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1672 {
1673     var result = new ulong[5];
1674     TryStepLeft(startLink, leftLink, result, 0);
1675     Links.Each(startLink, Constants.Any, couple =>
1676     {
1677         if (couple != startLink)
1678         {
1679             if (TryStepLeft(couple, leftLink, result, 2))
1680             {
1681                 return false;
1682             }
1683         }
1684         return true;
1685     });

```

```

1685         if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1686         {
1687             result[4] = leftLink;
1688         }
1689         return result;
1690     }
1691
1692     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1693     public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1694     {
1695         var added = 0;
1696         Links.Each(Constants.Any, startLink, couple =>
1697         {
1698             if (couple != startLink)
1699             {
1700                 var coupleSource = Links.GetSource(couple);
1701                 if (coupleSource == leftLink)
1702                 {
1703                     result[offset] = couple;
1704                     if (++added == 2)
1705                     {
1706                         return false;
1707                     }
1708                 }
1709                 else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1710                     ↪ == Net.And &&
1711                 {
1712                     result[offset + 1] = couple;
1713                     if (++added == 2)
1714                     {
1715                         return false;
1716                     }
1717                 }
1718             }
1719             return true;
1720         });
1721         return added > 0;
1722     }
1723 #endregion
1724
1725 #region Walkers
1726
1727 public class PatternMatcher : RightSequenceWalker<ulong>
1728 {
1729     private readonly Sequences _sequences;
1730     private readonly ulong[] _patternSequence;
1731     private readonly HashSet<LinkIndex> _linksInSequence;
1732     private readonly HashSet<LinkIndex> _results;
1733
1734     #region Pattern Match
1735
1736     enum PatternBlockType
1737     {
1738         Undefined,
1739         Gap,
1740         Elements
1741     }
1742
1743     struct PatternBlock
1744     {
1745         public PatternBlockType Type;
1746         public long Start;
1747         public long Stop;
1748     }
1749
1750     private readonly List<PatternBlock> _pattern;
1751     private int _patternPosition;
1752     private long _sequencePosition;
1753
1754     #endregion
1755
1756     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1757     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1758         ↪ HashSet<LinkIndex> results)
1759         : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1760     {
1761         _sequences = sequences;
1762         _patternSequence = patternSequence;
1763         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1764             ↪ _sequences.Constants.Any && x != ZeroOrMany));

```

```

1763         _results = results;
1764         _pattern = CreateDetailedPattern();
1765     }
1766
1767     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1768     protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
        ↳ base.IsElement(link);
1769
1770     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1771     public bool PatternMatch(LinkIndex sequenceToMatch)
1772     {
1773         _patternPosition = 0;
1774         _sequencePosition = 0;
1775         foreach (var part in Walk(sequenceToMatch))
1776         {
1777             if (!PatternMatchCore(part))
1778             {
1779                 break;
1780             }
1781         }
1782         return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
        ↳ - 1 && _pattern[_patternPosition].Start == 0);
1783     }
1784
1785     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1786     private List<PatternBlock> CreateDetailedPattern()
1787     {
1788         var pattern = new List<PatternBlock>();
1789         var patternBlock = new PatternBlock();
1790         for (var i = 0; i < _patternSequence.Length; i++)
1791         {
1792             if (patternBlock.Type == PatternBlockType.Undefined)
1793             {
1794                 if (_patternSequence[i] == _sequences.Constants.Any)
1795                 {
1796                     patternBlock.Type = PatternBlockType.Gap;
1797                     patternBlock.Start = 1;
1798                     patternBlock.Stop = 1;
1799                 }
1800                 else if (_patternSequence[i] == ZeroOrMany)
1801                 {
1802                     patternBlock.Type = PatternBlockType.Gap;
1803                     patternBlock.Start = 0;
1804                     patternBlock.Stop = long.MaxValue;
1805                 }
1806                 else
1807                 {
1808                     patternBlock.Type = PatternBlockType.Elements;
1809                     patternBlock.Start = i;
1810                     patternBlock.Stop = i;
1811                 }
1812             }
1813             else if (patternBlock.Type == PatternBlockType.Elements)
1814             {
1815                 if (_patternSequence[i] == _sequences.Constants.Any)
1816                 {
1817                     pattern.Add(patternBlock);
1818                     patternBlock = new PatternBlock
1819                     {
1820                         Type = PatternBlockType.Gap,
1821                         Start = 1,
1822                         Stop = 1
1823                     };
1824                 }
1825                 else if (_patternSequence[i] == ZeroOrMany)
1826                 {
1827                     pattern.Add(patternBlock);
1828                     patternBlock = new PatternBlock
1829                     {
1830                         Type = PatternBlockType.Gap,
1831                         Start = 0,
1832                         Stop = long.MaxValue
1833                     };
1834                 }
1835                 else
1836                 {
1837                     patternBlock.Stop = i;
1838                 }
1839             }
1840             else // patternBlock.Type == PatternBlockType.Gap

```



```

1841         {
1842             if (_patternSequence[i] == _sequences.Constants.Any)
1843             {
1844                 patternBlock.Start++;
1845                 if (patternBlock.Stop < patternBlock.Start)
1846                 {
1847                     patternBlock.Stop = patternBlock.Start;
1848                 }
1849             }
1850             else if (_patternSequence[i] == ZeroOrMany)
1851             {
1852                 patternBlock.Stop = long.MaxValue;
1853             }
1854             else
1855             {
1856                 pattern.Add(patternBlock);
1857                 patternBlock = new PatternBlock
1858                 {
1859                     Type = PatternBlockType.Elements,
1860                     Start = i,
1861                     Stop = i
1862                 };
1863             }
1864         }
1865     }
1866     if (patternBlock.Type != PatternBlockType.Undefined)
1867     {
1868         pattern.Add(patternBlock);
1869     }
1870     return pattern;
1871 }
1872
1873 // match: search for regexp anywhere in text
1874 //int match(char* regexp, char* text)
1875 //{
1876 //    do
1877 //    {
1878 //    } while (*text++ != '\0');
1879 //    return 0;
1880 //}
1881
1882 // matchhere: search for regexp at beginning of text
1883 //int matchhere(char* regexp, char* text)
1884 //{
1885 //    if (regexp[0] == '\0')
1886 //        return 1;
1887 //    if (regexp[1] == '*')
1888 //        return matchstar(regexp[0], regexp + 2, text);
1889 //    if (regexp[0] == '$' && regexp[1] == '\0')
1890 //        return *text == '\0';
1891 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1892 //        return matchhere(regexp + 1, text + 1);
1893 //    return 0;
1894 //}
1895
1896 // matchstar: search for c*regexp at beginning of text
1897 //int matchstar(int c, char* regexp, char* text)
1898 //{
1899 //    do
1900 //    {
1901 //        /* a * matches zero or more instances */
1902 //        if (matchhere(regexp, text))
1903 //            return 1;
1904 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1905 //    return 0;
1906 //}
1907
1908 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1909 //    long maximumGap)
1910 //{
1911 //    mininumGap = 0;
1912 //    maximumGap = 0;
1913 //    element = 0;
1914 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1915 //    {
1916 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1917 //            mininumGap++;
1918 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1919 //            maximumGap = long.MaxValue;
1920 //        else

```

```

1919         //          break;
1920     //    }
1921
1922     //    if (maximumGap < mininumGap)
1923     //        maximumGap = mininumGap;
1924     //}
1925
1926     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1927     private bool PatternMatchCore(LinkIndex element)
1928     {
1929         if (_patternPosition >= _pattern.Count)
1930         {
1931             _patternPosition = -2;
1932             return false;
1933         }
1934         var currentPatternBlock = _pattern[_patternPosition];
1935         if (currentPatternBlock.Type == PatternBlockType.Gap)
1936         {
1937             //var currentMatchingBlockLength = (_sequencePosition -
1938             ↪ _lastMatchedBlockPosition);
1939             if (_sequencePosition < currentPatternBlock.Start)
1940             {
1941                 _sequencePosition++;
1942                 return true; // Двигаемся дальше
1943             }
1944             // Это последний блок
1945             if (_pattern.Count == _patternPosition + 1)
1946             {
1947                 _patternPosition++;
1948                 _sequencePosition = 0;
1949                 return false; // Полное соответствие
1950             }
1951             else
1952             {
1953                 if (_sequencePosition > currentPatternBlock.Stop)
1954                 {
1955                     return false; // Соответствие невозможно
1956                 }
1957                 var nextPatternBlock = _pattern[_patternPosition + 1];
1958                 if (_patternSequence[nextPatternBlock.Start] == element)
1959                 {
1960                     if (nextPatternBlock.Start < nextPatternBlock.Stop)
1961                     {
1962                         _patternPosition++;
1963                         _sequencePosition = 1;
1964                     }
1965                     else
1966                     {
1967                         _patternPosition += 2;
1968                         _sequencePosition = 0;
1969                     }
1970                 }
1971             }
1972         }
1973         else // currentPatternBlock.Type == PatternBlockType.Elements
1974         {
1975             var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1976             if (_patternSequence[patternElementPosition] != element)
1977             {
1978                 return false; // Соответствие невозможно
1979             }
1980             if (patternElementPosition == currentPatternBlock.Stop)
1981             {
1982                 _patternPosition++;
1983                 _sequencePosition = 0;
1984             }
1985             else
1986             {
1987                 _sequencePosition++;
1988             }
1989         }
1990         return true;
1991         //if (_patternSequence[_patternPosition] != element)
1992         //    return false;
1993         //else
1994         //{
1995             //    _sequencePosition++;
1996             //    _patternPosition++;
1997             //    return true;
1998         //}

```

```

1997         //}
1998         //}
1999         //if (_filterPosition == _patternSequence.Length)
2000         //{
2001             //    _filterPosition = -2; // Длиннее чем нужно
2002             //    return false;
2003         //}
2004         //if (element != _patternSequence[_filterPosition])
2005         //{
2006             //    _filterPosition = -1;
2007             //    return false; // Начинается иначе
2008         //}
2009         //_filterPosition++;
2010         //if (_filterPosition == (_patternSequence.Length - 1))
2011             //    return false;
2012         //if (_filterPosition >= 0)
2013         //{
2014             //    if (element == _patternSequence[_filterPosition + 1])
2015                 //_filterPosition++;
2016             //    else
2017                 //    return false;
2018         //}
2019         //if (_filterPosition < 0)
2020         //{
2021             //    if (element == _patternSequence[0])
2022                 //_filterPosition = 0;
2023             //}
2024     }
2025
2026     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2027     public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
2028     {
2029         foreach (var sequenceToMatch in sequencesToMatch)
2030         {
2031             if (PatternMatch(sequenceToMatch))
2032             {
2033                 _results.Add(sequenceToMatch);
2034             }
2035         }
2036     }
2037 }
2038
2039 #endregion
2040 }
2041 }

```

## 1.82 ./csharp/Platform.Data.Doublets/Sequences/Sequences.cs

```

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;
11
12 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     /// <summary>
17     /// Представляет коллекцию последовательностей связей.
18     /// </summary>
19     /// <remarks>
20     /// Обязательно реализовать атомарность каждого публичного метода.
21     ///
22     /// TODO:
23     ///
24     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
25     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
26     /// вместе, все числа вместе и т.п.
27     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
28     /// графа)
29     ///
30     /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
31     /// ограничитель на то, что является последовательностью, а что нет,
32     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
33     /// 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
43  /// ↳ способами.
44  ///
45  /// Можно ли как-то сделать один общий интерфейс
46  ///
47  /// Блокчейн и/или гит для распределённой записи транзакций.
48  ///
49  /// </remarks>
50  public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
51  {
52  /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
53  /// ↳ связей.</summary>
54  public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
55
56  public SequencesOptions<LinkIndex> Options { get; }
57  public SynchronizedLinks<LinkIndex> Links { get; }
58  private readonly ISynchronization _sync;
59
60  public LinksConstants<LinkIndex> Constants { get; }
61
62  [MethodImpl(MethodImplOptions.AggressiveInlining)]
63  public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
64  {
65      Links = links;
66      _sync = links.SyncRoot;
67      Options = options;
68      Options.ValidateOptions();
69      Options.InitOptions(Links);
70      Constants = links.Constants;
71  }
72
73  [MethodImpl(MethodImplOptions.AggressiveInlining)]
74  public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
75  ↳ SequencesOptions<LinkIndex>()) { }
76
77  [MethodImpl(MethodImplOptions.AggressiveInlining)]
78  public bool IsSequence(LinkIndex sequence)
79  {
80      return _sync.ExecuteReadOperation(() =>
81      {
82          if (Options.UseSequenceMarker)
83          {
84              return Options.MarkedSequenceMatcher.IsMatched(sequence);
85          }
86          return !Links.Unsync.IsPartialPoint(sequence);
87      });
88  }
89
90  [MethodImpl(MethodImplOptions.AggressiveInlining)]
91  private LinkIndex GetSequenceByElements(LinkIndex sequence)
92  {
93      if (Options.UseSequenceMarker)
94      {
95          return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
96      }
97      return sequence;
98  }
99
100  [MethodImpl(MethodImplOptions.AggressiveInlining)]
101  private LinkIndex GetSequenceElements(LinkIndex sequence)
102  {
103      if (Options.UseSequenceMarker)
104      {
105          var linkContents = new Link<ulong>(Links.GetLink(sequence));

```

```

104         if (linkContents.Source == Options.SequenceMarkerLink)
105         {
106             return linkContents.Target;
107         }
108         if (linkContents.Target == Options.SequenceMarkerLink)
109         {
110             return linkContents.Source;
111         }
112     }
113     return sequence;
114 }
115
116 #region Count
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public LinkIndex Count(ICollection<LinkIndex> restrictions)
120 {
121     if (restrictions.IsNullOrEmpty())
122     {
123         return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
124     }
125     if (restrictions.Count == 1) // Первая связь это адрес
126     {
127         var sequenceIndex = restrictions[0];
128         if (sequenceIndex == Constants.Null)
129         {
130             return 0;
131         }
132         if (sequenceIndex == Constants.Any)
133         {
134             return Count(null);
135         }
136         if (Options.UseSequenceMarker)
137         {
138             return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
139         }
140         return Links.Exists(sequenceIndex) ? 1UL : 0;
141     }
142     throw new NotImplementedException();
143 }
144
145 [MethodImpl(MethodImplOptions.AggressiveInlining)]
146 private LinkIndex CountUsages(params LinkIndex[] restrictions)
147 {
148     if (restrictions.Length == 0)
149     {
150         return 0;
151     }
152     if (restrictions.Length == 1) // Первая связь это адрес
153     {
154         if (restrictions[0] == Constants.Null)
155         {
156             return 0;
157         }
158         var any = Constants.Any;
159         if (Options.UseSequenceMarker)
160         {
161             var elementsLink = GetSequenceElements(restrictions[0]);
162             var sequenceLink = GetSequenceByElements(elementsLink);
163             if (sequenceLink != Constants.Null)
164             {
165                 return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
166                     ↪ 1;
167             }
168             return Links.Count(any, elementsLink);
169         }
170         return Links.Count(any, restrictions[0]);
171     }
172     throw new NotImplementedException();
173 }
174
175 #endregion
176
177 #region Create
178
179 [MethodImpl(MethodImplOptions.AggressiveInlining)]
180 public LinkIndex Create(ICollection<LinkIndex> restrictions)
181 {
182     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(ICollection<LinkIndex> restrictions)
194 {
195     LinkIndex[] sequence = restrictions.SkipFirst();
196     if (Options.UseIndex)
197     {
198         Options.Index.Add(sequence);
199     }
200     var sequenceRoot = default(LinkIndex);
201     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
202     {
203         var matches = Each(restrictions);
204         if (matches.Count > 0)
205         {
206             sequenceRoot = matches[0];
207         }
208     }
209     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
210     {
211         return CompactCore(sequence);
212     }
213     if (sequenceRoot == default)
214     {
215         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
216     }
217     if (Options.UseSequenceMarker)
218     {
219         return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
220     }
221     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
222 }
223
224 #endregion
225
226 #region Each
227
228 [MethodImpl(MethodImplOptions.AggressiveInlining)]
229 public List<LinkIndex> Each(ICollection<LinkIndex> sequence)
230 {
231     var results = new List<LinkIndex>();
232     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
233     Each(filler.AddFirstAndReturnConstant, sequence);
234     return results;
235 }
236
237 [MethodImpl(MethodImplOptions.AggressiveInlining)]
238 public LinkIndex Each(Func<ICollection<LinkIndex>, LinkIndex> handler, ICollection<LinkIndex>
    ↪ restrictions)
239 {
240     return _sync.ExecuteReadOperation(() =>
241     {
242         if (restrictions.IsNullOrEmpty())
243         {
244             return Constants.Continue;
245         }
246         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
247         if (restrictions.Count == 1)
248         {
249             var link = restrictions[0];
250             var any = Constants.Any;
251             if (link == any)
252             {
253                 if (Options.UseSequenceMarker)
254                 {
255                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
256                         ↪ Options.SequenceMarkerLink, any));
257                 }
258                 else

```

```

258         {
259             return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
260                 ↪ any));
261         }
262     if (Options.UseSequenceMarker)
263     {
264         var sequenceLinkValues = Links.Unsync.GetLink(link);
265         if (sequenceLinkValues[Constants.SourcePart] ==
266             ↪ Options.SequenceMarkerLink)
267         {
268             link = sequenceLinkValues[Constants.TargetPart];
269         }
270         var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
271         sequence[0] = link;
272         return handler(sequence);
273     }
274     else if (restrictions.Count == 2)
275     {
276         throw new NotImplementedException();
277     }
278     else if (restrictions.Count == 3)
279     {
280         return Links.Unsync.Each(handler, restrictions);
281     }
282     else
283     {
284         var sequence = restrictions.SkipFirst();
285         if (Options.UseIndex && !Options.Index.MightContain(sequence))
286         {
287             return Constants.Break;
288         }
289         return EachCore(handler, sequence);
290     }
291     });
292 }
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
296     ↪ values)
297 {
298     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
299     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
300     ↪ Id.
301     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
302     ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
303     ↪ matcher.HandleFullMatched;
304     //if (sequence.Length >= 2)
305     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
306     {
307         return Constants.Break;
308     }
309     var last = values.Count - 2;
310     for (var i = 1; i < last; i++)
311     {
312         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
313             ↪ Constants.Continue)
314         {
315             return Constants.Break;
316         }
317     }
318     if (values.Count >= 3)
319     {
320         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
321             ↪ != Constants.Continue)
322         {
323             return Constants.Break;
324         }
325     }
326     return Constants.Continue;
327 }
328
329 [MethodImpl(MethodImplOptions.AggressiveInlining)]
330 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
331     ↪ left, LinkIndex right)
332 {
333     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 &&
416         ↪ !sequence.EqualTo(newSequence))
417     {
418         bestVariant = CompactCore(newSequence);
419     }
420     else
421     {
422         bestVariant = CreateCore(newSequence);
423     }
424     // TODO: Check all options only ones before loop execution
425     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
426     ↪ маркером,
427     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
428     ↪ можно получить имея только фактические последовательности.
429     foreach (var variant in Each(sequence))
430     {
431         if (variant != bestVariant)
432         {
433             UpdateOneCore(variant, bestVariant);
434         }
435     }
436     return bestVariant;
437 }
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
441 {
442     if (Options.UseGarbageCollection)
443     {
444         var sequenceElements = GetSequenceElements(sequence);
445         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
446         var sequenceLink = GetSequenceByElements(sequenceElements);
447         var newSequenceElements = GetSequenceElements(newSequence);
448         var newSequenceLink = GetSequenceByElements(newSequenceElements);
449         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
450         {
451             if (sequenceLink != Constants.Null)
452             {
453                 Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
454             }
455             Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
456         }
457         ClearGarbage(sequenceElementsContents.Source);
458         ClearGarbage(sequenceElementsContents.Target);
459     }
460     else
461     {
462         if (Options.UseSequenceMarker)
463         {
464             var sequenceElements = GetSequenceElements(sequence);
465             var sequenceLink = GetSequenceByElements(sequenceElements);
466             var newSequenceElements = GetSequenceElements(newSequence);
467             var newSequenceLink = GetSequenceByElements(newSequenceElements);
468             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
469             {
470                 if (sequenceLink != Constants.Null)
471                 {
472                     Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
473                 }
474                 Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
475             }
476         }
477     }
478 }

```

```

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(ICollection<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
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 private void DeleteOneCore(LinkIndex link)
504 {
505     if (Options.UseGarbageCollection)
506     {
507         var sequenceElements = GetSequenceElements(link);
508         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
509         var sequenceLink = GetSequenceByElements(sequenceElements);
510         if (Options.UseCascadeDelete || CountUsages(link) == 0)
511         {
512             if (sequenceLink != Constants.Null)
513             {
514                 Links.Unsync.Delete(sequenceLink);
515             }
516             Links.Unsync.Delete(link);
517         }
518         ClearGarbage(sequenceElementsContents.Source);
519         ClearGarbage(sequenceElementsContents.Target);
520     }
521     else
522     {
523         if (Options.UseSequenceMarker)
524         {
525             var sequenceElements = GetSequenceElements(link);
526             var sequenceLink = GetSequenceByElements(sequenceElements);
527             if (Options.UseCascadeDelete || CountUsages(link) == 0)
528             {
529                 if (sequenceLink != Constants.Null)
530                 {
531                     Links.Unsync.Delete(sequenceLink);
532                 }
533                 Links.Unsync.Delete(link);
534             }
535         }
536         else
537         {
538             if (Options.UseCascadeDelete || CountUsages(link) == 0)
539             {
540                 Links.Unsync.Delete(link);
541             }
542         }
543     }
544 }
545
546 #endregion
547
548 #region Compactification
549
550 [MethodImpl(MethodImplOptions.AggressiveInlining)]
551 public void CompactAll()

```

```

552 {
553     _sync.ExecuteWriteOperation(() =>
554     {
555         var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
556         for (int i = 0; i < sequences.Count; i++)
557         {
558             var sequence = this.ToList(sequences[i]);
559             Compact(sequence.ShiftRight());
560         }
561     });
562 }
563
564 /// <remarks>
565 /// bestVariant можно выбирать по максимальному числу использований,
566 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
567 /// гарантировать его использование в других местах).
568 ///
569 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
570 /// </remarks>
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 public LinkIndex Compact(ICollection<LinkIndex> sequence)
573 {
574     return _sync.ExecuteWriteOperation(() =>
575     {
576         if (sequence.IsNullOrEmpty())
577         {
578             return Constants.Null;
579         }
580         Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
581         return CompactCore(sequence);
582     });
583 }
584
585 [MethodImpl(MethodImplOptions.AggressiveInlining)]
586 private LinkIndex CompactCore(ICollection<LinkIndex> sequence) => UpdateCore(sequence,
587     ↪ sequence);
588
589 #endregion
590
591 #region Garbage Collection
592
593 /// <remarks>
594 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
595 ↪ определить извне или в унаследованном классе
596 /// </remarks>
597 [MethodImpl(MethodImplOptions.AggressiveInlining)]
598 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
599     ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
600
601 [MethodImpl(MethodImplOptions.AggressiveInlining)]
602 private void ClearGarbage(LinkIndex link)
603 {
604     if (IsGarbage(link))
605     {
606         var contents = new Link<ulong>(Links.GetLink(link));
607         Links.Unsync.Delete(link);
608         ClearGarbage(contents.Source);
609         ClearGarbage(contents.Target);
610     }
611 }
612
613 #endregion
614
615 #region Walkers
616
617 [MethodImpl(MethodImplOptions.AggressiveInlining)]
618 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
619 {
620     return _sync.ExecuteReadOperation(() =>
621     {
622         var links = Links.Unsync;
623         foreach (var part in Options.Walker.Walk(sequence))
624         {
625             if (!handler(part))
626             {
627                 return false;

```

```

628     });
629 }
630
631 public class Matcher : RightSequenceWalker<LinkIndex>
632 {
633     private readonly Sequences _sequences;
634     private readonly IList<LinkIndex> _patternSequence;
635     private readonly HashSet<LinkIndex> _linksInSequence;
636     private readonly HashSet<LinkIndex> _results;
637     private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
638     private readonly HashSet<LinkIndex> _readAsElements;
639     private int _filterPosition;
640
641     [MethodImpl(MethodImplOptions.AggressiveInlining)]
642     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
        ↳ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
        ↳ HashSet<LinkIndex> readAsElements = null)
        : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
643     {
644         _sequences = sequences;
645         _patternSequence = patternSequence;
646         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
647             ↳ _links.Constants.Any && x != ZeroOrMany));
648         _results = results;
649         _stopableHandler = stopableHandler;
650         _readAsElements = readAsElements;
651     }
652
653     [MethodImpl(MethodImplOptions.AggressiveInlining)]
654     protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
        ↳ (_readAsElements != null && _readAsElements.Contains(link)) ||
        ↳ _linksInSequence.Contains(link);
655
656     [MethodImpl(MethodImplOptions.AggressiveInlining)]
657     public bool FullMatch(LinkIndex sequenceToMatch)
658     {
659         _filterPosition = 0;
660         foreach (var part in Walk(sequenceToMatch))
661         {
662             if (!FullMatchCore(part))
663             {
664                 break;
665             }
666         }
667         return _filterPosition == _patternSequence.Count;
668     }
669
670     [MethodImpl(MethodImplOptions.AggressiveInlining)]
671     private bool FullMatchCore(LinkIndex element)
672     {
673         if (_filterPosition == _patternSequence.Count)
674         {
675             _filterPosition = -2; // Длиннее чем нужно
676             return false;
677         }
678         if (_patternSequence[_filterPosition] != _links.Constants.Any
679             && element != _patternSequence[_filterPosition])
680         {
681             _filterPosition = -1;
682             return false; // Начинается/Продолжается иначе
683         }
684         _filterPosition++;
685         return true;
686     }
687
688     [MethodImpl(MethodImplOptions.AggressiveInlining)]
689     public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
690     {
691         var sequenceToMatch = restrictions[_links.Constants.IndexPart];
692         if (FullMatch(sequenceToMatch))
693         {
694             _results.Add(sequenceToMatch);
695         }
696     }
697
698     [MethodImpl(MethodImplOptions.AggressiveInlining)]
699     public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
700     {
701         var sequenceToMatch = restrictions[_links.Constants.IndexPart];
702         if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))

```

```

703     {
704         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
705     }
706     return _links.Constants.Continue;
707 }
708
709 [MethodImpl(MethodImplOptions.AggressiveInlining)]
710 public LinkIndex HandleFullMatchedSequence(ICollection<LinkIndex> restrictions)
711 {
712     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
713     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
714     if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
715         ↪ _results.Add(sequenceToMatch))
716     {
717         return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
718     }
719     return _links.Constants.Continue;
720 }
721
722 /// <remarks>
723 /// TODO: Add support for LinksConstants.Any
724 /// </remarks>
725 [MethodImpl(MethodImplOptions.AggressiveInlining)]
726 public bool PartialMatch(LinkIndex sequenceToMatch)
727 {
728     _filterPosition = -1;
729     foreach (var part in Walk(sequenceToMatch))
730     {
731         if (!PartialMatchCore(part))
732         {
733             break;
734         }
735     }
736     return _filterPosition == _patternSequence.Count - 1;
737 }
738
739 [MethodImpl(MethodImplOptions.AggressiveInlining)]
740 private bool PartialMatchCore(LinkIndex element)
741 {
742     if (_filterPosition == (_patternSequence.Count - 1))
743     {
744         return false; // Нашлось
745     }
746     if (_filterPosition >= 0)
747     {
748         if (element == _patternSequence[_filterPosition + 1])
749         {
750             _filterPosition++;
751         }
752         else
753         {
754             _filterPosition = -1;
755         }
756     }
757     if (_filterPosition < 0)
758     {
759         if (element == _patternSequence[0])
760         {
761             _filterPosition = 0;
762         }
763     }
764     return true; // Ищем дальше
765 }
766
767 [MethodImpl(MethodImplOptions.AggressiveInlining)]
768 public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
769 {
770     if (PartialMatch(sequenceToMatch))
771     {
772         _results.Add(sequenceToMatch);
773     }
774 }
775
776 [MethodImpl(MethodImplOptions.AggressiveInlining)]
777 public LinkIndex HandlePartialMatched(ICollection<LinkIndex> restrictions)
778 {
779     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
780     if (PartialMatch(sequenceToMatch))
781     {

```

```

781         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
782     }
783     return _links.Constants.Continue;
784 }
785
786 [MethodImpl(MethodImplOptions.AggressiveInlining)]
787 public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
788 {
789     foreach (var sequenceToMatch in sequencesToMatch)
790     {
791         if (PartialMatch(sequenceToMatch))
792         {
793             _results.Add(sequenceToMatch);
794         }
795     }
796 }
797
798 [MethodImpl(MethodImplOptions.AggressiveInlining)]
799 public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
800 ↪ sequencesToMatch)
801 {
802     foreach (var sequenceToMatch in sequencesToMatch)
803     {
804         if (PartialMatch(sequenceToMatch))
805         {
806             _readAsElements.Add(sequenceToMatch);
807             _results.Add(sequenceToMatch);
808         }
809     }
810 }
811
812 #endregion
813 }
814 }

```

### 1.83 ./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[]>
13         ↪ groupedSequence)
14         {
15             var finalSequence = new TLink[groupedSequence.Count];
16             for (var i = 0; i < finalSequence.Length; i++)
17             {
18                 var part = groupedSequence[i];
19                 finalSequence[i] = part.Length == 1 ? part[0] :
20                 ↪ sequences.Create(part.ShiftRight());
21             }
22             return sequences.Create(finalSequence.ShiftRight());
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
27         {
28             var list = new List<TLink>();
29             var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
30             sequences.Each(filler.AddSkipFirstAndReturnConstant, new
31             ↪ LinkAddress<TLink>(sequence));
32             return list;
33         }
34     }
35 }

```

### 1.84 ./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         {
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             get;
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             set;
36         }
37
38         public bool UseCascadeDelete
39         {
40             [MethodImpl(MethodImplOptions.AggressiveInlining)]
41             get;
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             set;
44         }
45
46         public bool UseIndex
47         {
48             [MethodImpl(MethodImplOptions.AggressiveInlining)]
49             get;
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             set;
52         } // TODO: Update Index on sequence update/delete.
53
54         public bool UseSequenceMarker
55         {
56             [MethodImpl(MethodImplOptions.AggressiveInlining)]
57             get;
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]
59             set;
60         }
61
62         public bool UseCompression
63         {
64             [MethodImpl(MethodImplOptions.AggressiveInlining)]
65             get;
66             [MethodImpl(MethodImplOptions.AggressiveInlining)]
67             set;
68         }
69
70         public bool UseGarbageCollection
71         {
72             [MethodImpl(MethodImplOptions.AggressiveInlining)]
73             get;
74             [MethodImpl(MethodImplOptions.AggressiveInlining)]
75             set;
76         }
77
78         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
79         {
80             [MethodImpl(MethodImplOptions.AggressiveInlining)]
81             get;
82             [MethodImpl(MethodImplOptions.AggressiveInlining)]
83             set;

```

```

84     }
85
86     public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
87     {
88         [MethodImpl(MethodImplOptions.AggressiveInlining)]
89         get;
90         [MethodImpl(MethodImplOptions.AggressiveInlining)]
91         set;
92     }
93
94     public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher
95     {
96         [MethodImpl(MethodImplOptions.AggressiveInlining)]
97         get;
98         [MethodImpl(MethodImplOptions.AggressiveInlining)]
99         set;
100     }
101
102     public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
103     {
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         get;
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         set;
108     }
109
110     public ISequenceIndex<TLink> Index
111     {
112         [MethodImpl(MethodImplOptions.AggressiveInlining)]
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
156                             ↪ link.");
157                     }
158                 }
159             }
160             if (MarkedSequenceMatcher == null)
161             {
162                 MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
163                     ↪ 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
173                 ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
174                 ↪ MarkedSequenceMatcher);
175         }
176         else
177         {
178             totalSequenceSymbolFrequencyCounter = new
179                 ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
180         }
181         var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
182             ↪ totalSequenceSymbolFrequencyCounter);
183         var compressingConverter = new CompressingConverter<TLink>(links,
184             ↪ balancedVariantConverter, doubletFrequenciesCache);
185         LinksToSequenceConverter = compressingConverter;
186     }
187 }
188 else
189 {
190     if (LinksToSequenceConverter == null)
191     {
192         LinksToSequenceConverter = balancedVariantConverter;
193     }
194 }
195 if (UseIndex && Index == null)
196 {
197     Index = new SequenceIndex<TLink>(links);
198 }
199 if (Walker == null)
200 {
201     Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
202 }
203 }
204
205 [MethodImpl(MethodImplOptions.AggressiveInlining)]
206 public void ValidateOptions()
207 {
208     if (UseGarbageCollection && !UseSequenceMarker)
209     {
210         throw new NotSupportedException("To use garbage collection UseSequenceMarker
211             ↪ option must be on.");
212     }
213 }
214 }
215 }

```

#### 1.85 ./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 }

```

#### 1.86 ./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>
14             ↪ isElement) : base(links, stack, isElement) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
18             ↪ links.IsPartialPoint) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override TLink GetNextElementAfterPop(TLink element) =>
22             ↪ _links.GetSource(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override TLink GetNextElementAfterPush(TLink element) =>
26             ↪ _links.GetTarget(element);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override IEnumerable<TLink> WalkContents(TLink element)
30         {
31             var links = _links;
32             var parts = links.GetLink(element);
33             var start = links.Constants.SourcePart;
34             for (var i = parts.Count - 1; i >= start; i--)
35             {
36                 var part = parts[i];
37                 if (IsElement(part))
38                 {
39                     yield return part;
40                 }
41             }
42         }
43     }
44 }

```

#### 1.87 ./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.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 // #define USEARRAYPOOL
8 #if USEARRAYPOOL
9 using Platform.Collections;
10 #endif
11
12 namespace Platform.Data.Doublets.Sequences.Walkers
13 {
14     public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
15     {
16         private static readonly EqualityComparer<TLink> _equalityComparer =
17             ↪ EqualityComparer<TLink>.Default;
18
19         private readonly Func<TLink, bool> _isElement;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
23             ↪ base(links) => _isElement = isElement;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
27             ↪ _links.IsPartialPoint;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public TLink[] ToArray(TLink sequence)
34         {
35             var length = 1;
36             var array = new TLink[length];
37             array[0] = sequence;
38             if (_isElement(sequence))
39             {
40                 return array;
41             }
42             bool hasElements;

```

```

40         do
41         {
42             length *= 2;
43 #if USEARRAYPOOL
44             var nextArray = ArrayPool.Allocate<ulong>(length);
45 #else
46             var nextArray = new TLink[length];
47 #endif
48             hasElements = false;
49             for (var i = 0; i < array.Length; i++)
50             {
51                 var candidate = array[i];
52                 if (!_equalityComparer.Equals(array[i], default))
53                 {
54                     continue;
55                 }
56                 var doubletOffset = i * 2;
57                 if (!_isElement(candidate))
58                 {
59                     nextArray[doubletOffset] = candidate;
60                 }
61                 else
62                 {
63                     var links = _links;
64                     var link = links.GetLink(candidate);
65                     var linkSource = links.GetSource(link);
66                     var linkTarget = links.GetTarget(link);
67                     nextArray[doubletOffset] = linkSource;
68                     nextArray[doubletOffset + 1] = linkTarget;
69                     if (!hasElements)
70                     {
71                         hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
72                     }
73                 }
74             }
75 #if USEARRAYPOOL
76             if (array.Length > 1)
77             {
78                 ArrayPool.Free(array);
79             }
80 #endif
81             array = nextArray;
82         }
83         while (hasElements);
84         var filledElementsCount = CountFilledElements(array);
85         if (filledElementsCount == array.Length)
86         {
87             return array;
88         }
89         else
90         {
91             return CopyFilledElements(array, filledElementsCount);
92         }
93     }
94
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
97     {
98         var finalArray = new TLink[filledElementsCount];
99         for (int i = 0, j = 0; i < array.Length; i++)
100         {
101             if (!_equalityComparer.Equals(array[i], default))
102             {
103                 finalArray[j] = array[i];
104                 j++;
105             }
106         }
107 #if USEARRAYPOOL
108         ArrayPool.Free(array);
109 #endif
110         return finalArray;
111     }
112
113     [MethodImpl(MethodImplOptions.AggressiveInlining)]
114     private static int CountFilledElements(TLink[] array)
115     {
116         var count = 0;
117         for (var i = 0; i < array.Length; i++)
118         {

```

```

119         if (!_equalityComparer.Equals(array[i], default))
120         {
121             count++;
122         }
123     }
124     return count;
125 }
126 }
127 }

```

#### 1.88 ./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.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 RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
14             ↪ isElement) : base(links, stack, isElement) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
18             ↪ stack, links.IsPartialPoint) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override TLink GetNextElementAfterPop(TLink element) =>
22             ↪ _links.GetTarget(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override TLink GetNextElementAfterPush(TLink element) =>
26             ↪ _links.GetSource(element);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override IEnumerable<TLink> WalkContents(TLink element)
30         {
31             var parts = _links.GetLink(element);
32             for (var i = _links.Constants.SourcePart; i < parts.Count; i++)
33             {
34                 var part = parts[i];
35                 if (IsElement(part))
36                 {
37                     yield return part;
38                 }
39             }
40         }
41     }
42 }

```

#### 1.89 ./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.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 abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
11         ↪ ISequenceWalker<TLink>
12     {
13         private readonly IStack<TLink> _stack;
14         private readonly Func<TLink, bool> _isElement;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
18             ↪ isElement) : base(links)
19         {
20             _stack = stack;
21             _isElement = isElement;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

23     protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
24         ↪ stack, links.IsPartialPoint) { }
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     public IEnumerable<TLink> Walk(TLink sequence)
28     {
29         _stack.Clear();
30         var element = sequence;
31         if (IsElement(element))
32         {
33             yield return element;
34         }
35         else
36         {
37             while (true)
38             {
39                 if (IsElement(element))
40                 {
41                     if (_stack.IsEmpty)
42                     {
43                         break;
44                     }
45                     element = _stack.Pop();
46                     foreach (var output in WalkContents(element))
47                     {
48                         yield return output;
49                     }
50                     element = GetNextElementAfterPop(element);
51                 }
52                 else
53                 {
54                     _stack.Push(element);
55                     element = GetNextElementAfterPush(element);
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 }

```

## 1.90 ./csharp/Platform.Data.Doublets/SplitMemory/Generic/LinksSizeBalancedTreeMethodsBase.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using 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.SplitMemory.Generic
12 {
13     public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> :
14         ↪ SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
17             ↪ UncheckedConverter<TLink, long>.Default;
18
19         protected readonly TLink Break;
20         protected readonly TLink Continue;
21         protected readonly byte* LinksDataParts;
22         protected readonly byte* LinksIndexParts;
23         protected readonly byte* Header;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte*
27             ↪ linksDataParts, byte* linksIndexParts, byte* header)

```

```

25 {
26     LinksDataParts = linksDataParts;
27     LinksIndexParts = linksIndexParts;
28     Header = header;
29     Break = constants.Break;
30     Continue = constants.Continue;
31 }
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 protected abstract TLink GetTreeRoot(TLink link);
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected abstract TLink GetBasePartValue(TLink link);
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected abstract TLink GetKeyPartValue(TLink link);
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
    ↪ AsRef<RawLinkDataPart<TLink>>(LinksDataParts + (RawLinkDataPart<TLink>.SizeInBytes *
    ↪ _addressToInt64Converter.Convert(link)));
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
    ↪ ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
    ↪ (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
    ↪ LessThan(GetKeyPartValue(first), GetKeyPartValue(second));
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
    ↪ GreaterThan(GetKeyPartValue(first), GetKeyPartValue(second));
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
56 {
57     ref var link = ref GetLinkDataPartReference(linkIndex);
58     return new Link<TLink>(linkIndex, link.Source, link.Target);
59 }
60
61 public TLink this[TLink link, TLink index]
62 {
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     get
65     {
66         var root = GetTreeRoot(link);
67         if (GreaterOrEqualThan(index, GetSize(root)))
68         {
69             return Zero;
70         }
71         while (!EqualToZero(root))
72         {
73             var left = GetLeftOrDefault(root);
74             var leftSize = GetSizeOrZero(left);
75             if (LessThan(index, leftSize))
76             {
77                 root = left;
78                 continue;
79             }
80             if (AreEqual(index, leftSize))
81             {
82                 return root;
83             }
84             root = GetRightOrDefault(root);
85             index = Subtract(index, Increment(leftSize));
86         }
87         return Zero; // TODO: Impossible situation exception (only if tree structure
            ↪ broken)
88     }
89 }
90
91 /// <summary>
92 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
    ↪ (концом).
93 /// </summary>
94 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
95 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>

```

```

96     /// <returns>Индекс искомой связи.</returns>
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     public abstract TLink Search(TLink source, TLink target);
99
100    [MethodImpl(MethodImplOptions.AggressiveInlining)]
101    protected TLink SearchCore(TLink root, TLink key)
102    {
103        while (!EqualToZero(root))
104        {
105            var rootKey = GetKeyPartValue(root);
106            if (LessThan(key, rootKey)) // node.Key < root.Key
107            {
108                root = GetLeftOrDefault(root);
109            }
110            else if (GreaterThan(key, rootKey)) // node.Key > root.Key
111            {
112                root = GetRightOrDefault(root);
113            }
114            else // node.Key == root.Key
115            {
116                return root;
117            }
118        }
119        return Zero;
120    }
121
122    // TODO: Return indices range instead of references count
123    [MethodImpl(MethodImplOptions.AggressiveInlining)]
124    public TLink CountUsages(TLink link) => GetSizeOrZero(GetTreeRoot(link));
125
126    [MethodImpl(MethodImplOptions.AggressiveInlining)]
127    public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
128        ↪ EachUsageCore(@base, GetTreeRoot(@base), handler);
129
130    // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
131    ↪ low-level MSIL stack.
132    [MethodImpl(MethodImplOptions.AggressiveInlining)]
133    private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
134    {
135        var @continue = Continue;
136        if (EqualToZero(link))
137        {
138            return @continue;
139        }
140        var @break = Break;
141        if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
142        {
143            return @break;
144        }
145        if (AreEqual(handler(GetLinkValues(link)), @break))
146        {
147            return @break;
148        }
149        if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
150        {
151            return @break;
152        }
153        return @continue;
154    }
155
156    [MethodImpl(MethodImplOptions.AggressiveInlining)]
157    protected override void PrintNodeValue(TLink node, StringBuilder sb)
158    {
159        ref var link = ref GetLinkDataPartReference(node);
160        sb.Append(' ');
161        sb.Append(link.Source);
162        sb.Append('-');
163        sb.Append('>');
164        sb.Append(link.Target);
165    }
166 }

```

1.91 ./csharp/Platform.Data.Doublets.SplitMemory.Generic/LinksSourcesSizeBalancedTreeMethods.cs

```

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

```

```

6 {
7     public unsafe class LinksSourcesSizeBalancedTreeMethods<TLink> :
        ↳ LinksSizeBalancedTreeMethodsBase<TLink>
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public LinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte*
        ↳ linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
        ↳ linksDataParts, linksIndexParts, header) { }
11
12        [MethodImpl(MethodImplOptions.AggressiveInlining)]
13        protected override ref TLink GetLeftReference(TLink node) => ref
        ↳ GetLinkIndexPartReference(node).LeftAsSource;
14
15        [MethodImpl(MethodImplOptions.AggressiveInlining)]
16        protected override ref TLink GetRightReference(TLink node) => ref
        ↳ GetLinkIndexPartReference(node).RightAsSource;
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        protected override TLink GetLeft(TLink node) =>
        ↳ GetLinkIndexPartReference(node).LeftAsSource;
20
21        [MethodImpl(MethodImplOptions.AggressiveInlining)]
22        protected override TLink GetRight(TLink node) =>
        ↳ GetLinkIndexPartReference(node).RightAsSource;
23
24        [MethodImpl(MethodImplOptions.AggressiveInlining)]
25        protected override void SetLeft(TLink node, TLink left) =>
        ↳ GetLinkIndexPartReference(node).LeftAsSource = left;
26
27        [MethodImpl(MethodImplOptions.AggressiveInlining)]
28        protected override void SetRight(TLink node, TLink right) =>
        ↳ GetLinkIndexPartReference(node).RightAsSource = right;
29
30        [MethodImpl(MethodImplOptions.AggressiveInlining)]
31        protected override TLink GetSize(TLink node) =>
        ↳ GetLinkIndexPartReference(node).SizeAsSource;
32
33        [MethodImpl(MethodImplOptions.AggressiveInlining)]
34        protected override void SetSize(TLink node, TLink size) =>
        ↳ GetLinkIndexPartReference(node).SizeAsSource = size;
35
36        [MethodImpl(MethodImplOptions.AggressiveInlining)]
37        protected override TLink GetTreeRoot(TLink link) =>
        ↳ GetLinkIndexPartReference(link).RootAsSource;
38
39        [MethodImpl(MethodImplOptions.AggressiveInlining)]
40        protected override TLink GetBasePartValue(TLink link) =>
        ↳ GetLinkDataPartReference(link).Source;
41
42        [MethodImpl(MethodImplOptions.AggressiveInlining)]
43        protected override TLink GetKeyPartValue(TLink link) =>
        ↳ GetLinkDataPartReference(link).Target;
44
45        [MethodImpl(MethodImplOptions.AggressiveInlining)]
46        protected override void ClearNode(TLink node)
47        {
48            ref var link = ref GetLinkIndexPartReference(node);
49            link.LeftAsSource = Zero;
50            link.RightAsSource = Zero;
51            link.SizeAsSource = Zero;
52        }
53
54        public override TLink Search(TLink source, TLink target) =>
        ↳ SearchCore(GetTreeRoot(source), target);
55    }
56 }

```

## 1.92 ./csharp/Platform.Data.Doublets.SplitMemory.Generic/LinksTargetsSizeBalancedTreeMethods.cs

```

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

```



```

10     public LinksTargetsSizeBalancedTreeMethods(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
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 }

```

### 1.93 ./csharp/Platform.Data.Doublets/SplitMemory/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.SplitMemory.Generic
10 {
11     public unsafe class SplitMemoryLinks<TLink> : SplitMemoryLinksBase<TLink>
12     {
13         private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
14         private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
15         private byte* _header;
16         private byte* _linksDataParts;

```

```

17     private byte* _linksIndexParts;
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↳ indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↳ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
        ↳ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
        ↳ base(dataMemory, indexMemory, memoryReservationStep, constants)
27     {
28         _createSourceTreeMethods = () => new
        ↳ LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
        ↳ _linksIndexParts, _header);
29         _createTargetTreeMethods = () => new
        ↳ LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
        ↳ _linksIndexParts, _header);
30         Init(dataMemory, indexMemory, memoryReservationStep);
31     }
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override void SetPointers(IResizableDirectMemory dataMemory,
        ↳ IResizableDirectMemory indexMemory)
35     {
36         _linksDataParts = (byte*)dataMemory.Pointer;
37         _linksIndexParts = (byte*)indexMemory.Pointer;
38         _header = _linksIndexParts;
39         SourcesTreeMethods = _createSourceTreeMethods();
40         TargetsTreeMethods = _createTargetTreeMethods();
41         UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_linksDataParts, _header);
42     }
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override void ResetPointers()
46     {
47         base.ResetPointers();
48         _linksDataParts = null;
49         _linksIndexParts = null;
50         _header = null;
51     }
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override ref LinksHeaderIndexPart<TLink> GetHeaderReference() => ref
        ↳ AsRef<LinksHeaderIndexPart<TLink>>(_header);
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
        ↳ => ref AsRef<RawLinkDataPart<TLink>>(_linksDataParts + (LinkDataPartSizeInBytes *
        ↳ ConvertToInt64(linkIndex)));
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
        ↳ linkIndex) => ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
        ↳ (LinkIndexPartSizeInBytes * ConvertToInt64(linkIndex)));
61 }
62 }

```

#### 1.94 ./csharp/Platform.Data.Doublets/SplitMemory/Generic/SplitMemoryLinksBase.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Singletons;
6  using Platform.Converters;
7  using Platform.Numbers;
8  using Platform.Memory;
9  using Platform.Data.Exceptions;
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.SplitMemory.Generic
14 {
15     public abstract class SplitMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16     {

```

```

17 private static readonly EqualityComparer<TLink> _equalityComparer =
18     ↳ EqualityComparer<TLink>.Default;
19 private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
20 private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
21     ↳ UncheckedConverter<TLink, long>.Default;
22 private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
23     ↳ UncheckedConverter<long, TLink>.Default;
24
25 private static readonly TLink _zero = default;
26 private static readonly TLink _one = Arithmetic.Increment(_zero);
27
28 /// <summary>Возвращает размер одной связи в байтах.</summary>
29 /// <remarks>
30 /// Используется только во вне класса, не рекомендуется использовать внутри.
31 /// Так как во вне не обязательно будет доступен unsafe C#.
32 /// </remarks>
33 public static readonly long LinkDataPartSizeInBytes = RawLinkDataPart<TLink>.SizeInBytes;
34
35 public static readonly long LinkIndexPartSizeInBytes =
36     ↳ RawLinkIndexPart<TLink>.SizeInBytes;
37
38 public static readonly long LinkHeaderSizeInBytes =
39     ↳ LinksHeaderIndexPart<TLink>.SizeInBytes;
40
41 public static readonly long DefaultLinksSizeStep = 1 * 1024 * 1024;
42
43 protected readonly IResizableDirectMemory _dataMemory;
44 protected readonly IResizableDirectMemory _indexMemory;
45 protected readonly long _dataMemoryReservationStepInBytes;
46 protected readonly long _indexMemoryReservationStepInBytes;
47
48 protected ILinksTreeMethods<TLink> TargetsTreeMethods;
49 protected ILinksTreeMethods<TLink> SourcesTreeMethods;
50 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
51 // ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
52 // ↳ наличие связи внутри
53 protected ILinksListMethods<TLink> UnusedLinksListMethods;
54
55 /// <summary>
56 /// Возвращает общее число связей находящихся в хранилище.
57 /// </summary>
58 protected virtual TLink Total
59 {
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     get
62     {
63         ref var header = ref GetHeaderReference();
64         return Subtract(header.AllocatedLinks, header.FreeLinks);
65     }
66 }
67
68 public virtual LinksConstants<TLink> Constants
69 {
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     get;
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
76     ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants)
77 {
78     _dataMemory = dataMemory;
79     _indexMemory = indexMemory;
80     _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
81     _indexMemoryReservationStepInBytes = memoryReservationStep *
82     ↳ LinkIndexPartSizeInBytes;
83     Constants = constants;
84 }
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
88     ↳ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
89     ↳ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
93     ↳ indexMemory, long memoryReservationStep)
94 {
95     if (dataMemory.ReservedCapacity < memoryReservationStep)
96     {

```

```

85     dataMemory.ReservedCapacity = memoryReservationStep;
86 }
87 if (indexMemory.ReservedCapacity < memoryReservationStep)
88 {
89     indexMemory.ReservedCapacity = memoryReservationStep;
90 }
91 SetPointers(dataMemory, indexMemory);
92 ref var header = ref GetHeaderReference();
93 // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
94 // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
95 dataMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
    ↪ LinkDataPartSizeInBytes + LinkDataPartSizeInBytes; // First link is read only
    ↪ zero link.
96 indexMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
    ↪ LinkIndexPartSizeInBytes + LinkHeaderSizeInBytes;
97 // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
98 // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
99 header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
    ↪ LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
100 }
101
102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 public virtual TLink Count(IList<TLink> restrictions)
104 {
105     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
106     if (restrictions.Count == 0)
107     {
108         return Total;
109     }
110     var constants = Constants;
111     var any = constants.Any;
112     var index = restrictions[constants.IndexPart];
113     if (restrictions.Count == 1)
114     {
115         if (AreEqual(index, any))
116         {
117             return Total;
118         }
119         return Exists(index) ? GetOne() : GetZero();
120     }
121     if (restrictions.Count == 2)
122     {
123         var value = restrictions[1];
124         if (AreEqual(index, any))
125         {
126             if (AreEqual(value, any))
127             {
128                 return Total; // Any - как отсутствие ограничения
129             }
130             return Add(SourcesTreeMethods.CountUsages(value),
    ↪ TargetsTreeMethods.CountUsages(value));
131         }
132         else
133         {
134             if (!Exists(index))
135             {
136                 return GetZero();
137             }
138             if (AreEqual(value, any))
139             {
140                 return GetOne();
141             }
142             ref var storedLinkValue = ref GetLinkDataPartReference(index);
143             if (AreEqual(storedLinkValue.Source, value) ||
    ↪ AreEqual(storedLinkValue.Target, value))
144             {
145                 return GetOne();
146             }
147             return GetZero();
148         }
149     }
150     if (restrictions.Count == 3)
151     {
152         var source = restrictions[constants.SourcePart];
153         var target = restrictions[constants.TargetPart];
154         if (AreEqual(index, any))
155         {
156             if (AreEqual(source, any) && AreEqual(target, any))

```

```

157         {
158             return Total;
159         }
160         else if (AreEqual(source, any))
161         {
162             return TargetsTreeMethods.CountUsages(target);
163         }
164         else if (AreEqual(target, any))
165         {
166             return SourcesTreeMethods.CountUsages(source);
167         }
168         else //if(source != Any && target != Any)
169         {
170             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
171             var link = SourcesTreeMethods.Search(source, target);
172             return AreEqual(link, constants.Null) ? GetZero() : GetOne();
173         }
174     }
175     else
176     {
177         if (!Exists(index))
178         {
179             return GetZero();
180         }
181         if (AreEqual(source, any) && AreEqual(target, any))
182         {
183             return GetOne();
184         }
185         ref var storedLinkValue = ref GetLinkDataPartReference(index);
186         if (!AreEqual(source, any) && !AreEqual(target, any))
187         {
188             if (AreEqual(storedLinkValue.Source, source) &&
189                 ⇨ AreEqual(storedLinkValue.Target, target))
190             {
191                 return GetOne();
192             }
193             return GetZero();
194         }
195         var value = default(TLink);
196         if (AreEqual(source, any))
197         {
198             value = target;
199         }
200         if (AreEqual(target, any))
201         {
202             value = source;
203         }
204         if (AreEqual(storedLinkValue.Source, value) ||
205             ⇨ AreEqual(storedLinkValue.Target, value))
206         {
207             return GetOne();
208         }
209         return GetZero();
210     }
211 }
212
213 [MethodImpl(MethodImplOptions.AggressiveInlining)]
214 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
215 {
216     var constants = Constants;
217     var @break = constants.Break;
218     if (restrictions.Count == 0)
219     {
220         for (var link = GetOne(); LessOrEqualThan(link,
221             ⇨ GetHeaderReference().AllocatedLinks); link = Increment(link))
222         {
223             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
224             {
225                 return @break;
226             }
227         }
228         return @break;
229     }
230     var @continue = constants.Continue;
231     var any = constants.Any;

```

```

231 var index = restrictions[constants.IndexPart];
232 if (restrictions.Count == 1)
233 {
234     if (AreEqual(index, any))
235     {
236         return Each(handler, Array.Empty<TLink>());
237     }
238     if (!Exists(index))
239     {
240         return @continue;
241     }
242     return handler(GetLinkStruct(index));
243 }
244 if (restrictions.Count == 2)
245 {
246     var value = restrictions[1];
247     if (AreEqual(index, any))
248     {
249         if (AreEqual(value, any))
250         {
251             return Each(handler, Array.Empty<TLink>());
252         }
253         if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
254         {
255             return @break;
256         }
257         return Each(handler, new Link<TLink>(index, any, value));
258     }
259     else
260     {
261         if (!Exists(index))
262         {
263             return @continue;
264         }
265         if (AreEqual(value, any))
266         {
267             return handler(GetLinkStruct(index));
268         }
269         ref var storedLinkValue = ref GetLinkDataPartReference(index);
270         if (AreEqual(storedLinkValue.Source, value) ||
271             AreEqual(storedLinkValue.Target, value))
272         {
273             return handler(GetLinkStruct(index));
274         }
275         return @continue;
276     }
277 }
278 if (restrictions.Count == 3)
279 {
280     var source = restrictions[constants.SourcePart];
281     var target = restrictions[constants.TargetPart];
282     if (AreEqual(index, any))
283     {
284         if (AreEqual(source, any) && AreEqual(target, any))
285         {
286             return Each(handler, Array.Empty<TLink>());
287         }
288         else if (AreEqual(source, any))
289         {
290             return TargetsTreeMethods.EachUsage(target, handler);
291         }
292         else if (AreEqual(target, any))
293         {
294             return SourcesTreeMethods.EachUsage(source, handler);
295         }
296         else //if(source != Any && target != Any)
297         {
298             var link = SourcesTreeMethods.Search(source, target);
299             return AreEqual(link, constants.Null) ? @continue :
300                 ↪ handler(GetLinkStruct(link));
301         }
302     }
303     else
304     {
305         if (!Exists(index))
306         {
307             return @continue;
308         }

```

```

308         if (AreEqual(source, any) && AreEqual(target, any))
309         {
310             return handler(GetLinkStruct(index));
311         }
312         ref var storedLinkValue = ref GetLinkDataPartReference(index);
313         if (!AreEqual(source, any) && !AreEqual(target, any))
314         {
315             if (AreEqual(storedLinkValue.Source, source) &&
316                 AreEqual(storedLinkValue.Target, target))
317             {
318                 return handler(GetLinkStruct(index));
319             }
320             return @continue;
321         }
322         var value = default(TLink);
323         if (AreEqual(source, any))
324         {
325             value = target;
326         }
327         if (AreEqual(target, any))
328         {
329             value = source;
330         }
331         if (AreEqual(storedLinkValue.Source, value) ||
332             AreEqual(storedLinkValue.Target, value))
333         {
334             return handler(GetLinkStruct(index));
335         }
336         return @continue;
337     }
338 }
339 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
340 }
341
342 /// <remarks>
343 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
344 /// </remarks>
345 [MethodImpl(MethodImplOptions.AggressiveInlining)]
346 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
347 {
348     var constants = Constants;
349     var @null = constants.Null;
350     var linkIndex = restrictions[constants.IndexPart];
351     ref var link = ref GetLinkDataPartReference(linkIndex);
352     ref var header = ref GetHeaderReference();
353     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
354     if (!AreEqual(link.Source, @null))
355     {
356         SourcesTreeMethods.Detach(ref
    ↳ GetLinkIndexPartReference(link.Source).RootAsSource, linkIndex);
357     }
358     if (!AreEqual(link.Target, @null))
359     {
360         TargetsTreeMethods.Detach(ref
    ↳ GetLinkIndexPartReference(link.Target).RootAsTarget, linkIndex);
361     }
362     link.Source = substitution[constants.SourcePart];
363     link.Target = substitution[constants.TargetPart];
364     if (!AreEqual(link.Source, @null))
365     {
366         SourcesTreeMethods.Attach(ref
    ↳ GetLinkIndexPartReference(link.Source).RootAsSource, linkIndex);
367     }
368     if (!AreEqual(link.Target, @null))
369     {
370         TargetsTreeMethods.Attach(ref
    ↳ GetLinkIndexPartReference(link.Target).RootAsTarget, linkIndex);
371     }
372     return linkIndex;
373 }
374
375 /// <remarks>
376 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
    ↳ пространство
377 /// </remarks>

```

```

378 [MethodImpl(MethodImplOptions.AggressiveInlining)]
379 public virtual TLink Create(IList<TLink> restrictions)
380 {
381     ref var header = ref GetHeaderReference();
382     var freeLink = header.FirstFreeLink;
383     if (!AreEqual(freeLink, Constants.Null))
384     {
385         UnusedLinksListMethods.Detach(freeLink);
386     }
387     else
388     {
389         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
390         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
391         {
392             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
393         }
394         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
395         {
396             _dataMemory.ReservedCapacity += _dataMemory.ReservationStepInBytes;
397             _indexMemory.ReservedCapacity += _indexMemory.ReservationStepInBytes;
398             SetPointers(_dataMemory, _indexMemory);
399             header = ref GetHeaderReference();
400             header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
401                 ↳ LinkDataPartSizeInBytes);
402         }
403         header.AllocatedLinks = Increment(header.AllocatedLinks);
404         _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
405         _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
406         freeLink = header.AllocatedLinks;
407     }
408     return freeLink;
409 }
410 [MethodImpl(MethodImplOptions.AggressiveInlining)]
411 public virtual void Delete(IList<TLink> restrictions)
412 {
413     ref var header = ref GetHeaderReference();
414     var link = restrictions[Constants.IndexPart];
415     if (LessThan(link, header.AllocatedLinks)
416     {
417         UnusedLinksListMethods.AttachAsFirst(link);
418     }
419     else if (AreEqual(link, header.AllocatedLinks))
420     {
421         header.AllocatedLinks = Decrement(header.AllocatedLinks);
422         _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
423         _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
424         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
425         ↳ пока не дойдём до первой существующей связи
426         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
427         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
428             ↳ IsUnusedLink(header.AllocatedLinks))
429         {
430             UnusedLinksListMethods.Detach(header.AllocatedLinks);
431             header.AllocatedLinks = Decrement(header.AllocatedLinks);
432             _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
433             _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
434         }
435     }
436 }
437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
438 public IList<TLink> GetLinkStruct(TLink linkIndex)
439 {
440     ref var link = ref GetLinkDataPartReference(linkIndex);
441     return new Link<TLink>(linkIndex, link.Source, link.Target);
442 }
443 /// <remarks>
444 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
445 ↳ адрес реально поменялся
446 ///
447 /// Указатель this.links может быть в том же месте,
448 /// так как 0-я связь не используется и имеет такой же размер как Header,
449 /// поэтому header размещается в том же месте, что и 0-я связь
450 /// </remarks>
451 [MethodImpl(MethodImplOptions.AggressiveInlining)]
452 protected abstract void SetPointers(IResizableDirectMemory dataMemory,
453     ↳ IResizableDirectMemory indexMemory);

```



```

452 [MethodImpl(MethodImplOptions.AggressiveInlining)]
453 protected virtual void ResetPointers()
454 {
455     SourcesTreeMethods = null;
456     TargetsTreeMethods = null;
457     UnusedLinksListMethods = null;
458 }
459
460 [MethodImpl(MethodImplOptions.AggressiveInlining)]
461 protected abstract ref LinksHeaderIndexPart<TLink> GetHeaderReference();
462
463 [MethodImpl(MethodImplOptions.AggressiveInlining)]
464 protected abstract ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex);
465
466 [MethodImpl(MethodImplOptions.AggressiveInlining)]
467 protected abstract ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
468     ↪ linkIndex);
469
470 [MethodImpl(MethodImplOptions.AggressiveInlining)]
471 protected virtual bool Exists(TLink link)
472     => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
473     && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
474     && !IsUnusedLink(link);
475
476 [MethodImpl(MethodImplOptions.AggressiveInlining)]
477 protected virtual bool IsUnusedLink(TLink linkIndex)
478 {
479     if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
480         ↪ is not needed
481     {
482         // TODO: Reduce access to memory in different location (should be enough to use
483         ↪ just linkIndexPart)
484         ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
485         ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
486         return AreEqual(linkIndexPart.SizeAsSource, default) &&
487             ↪ !AreEqual(linkDataPart.Source, default);
488     }
489     else
490     {
491         return true;
492     }
493 }
494
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 protected virtual TLink GetOne() => _one;
497
498 [MethodImpl(MethodImplOptions.AggressiveInlining)]
499 protected virtual TLink GetZero() => default;
500
501 [MethodImpl(MethodImplOptions.AggressiveInlining)]
502 protected virtual bool AreEqual(TLink first, TLink second) =>
503     ↪ _equalityComparer.Equals(first, second);
504
505 [MethodImpl(MethodImplOptions.AggressiveInlining)]
506 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
507     ↪ second) < 0;
508
509 [MethodImpl(MethodImplOptions.AggressiveInlining)]
510 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
511     ↪ _comparer.Compare(first, second) <= 0;
512
513 [MethodImpl(MethodImplOptions.AggressiveInlining)]
514 protected virtual bool GreaterThan(TLink first, TLink second) =>
515     ↪ _comparer.Compare(first, second) > 0;
516
517 [MethodImpl(MethodImplOptions.AggressiveInlining)]
518 protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
519     ↪ _comparer.Compare(first, second) >= 0;
520
521 [MethodImpl(MethodImplOptions.AggressiveInlining)]
522 protected virtual long ConvertToInt64(TLink value) =>
523     ↪ _addressToInt64Converter.Convert(value);
524
525 [MethodImpl(MethodImplOptions.AggressiveInlining)]
526 protected virtual TLink ConvertToAddress(long value) =>
527     ↪ _int64ToAddressConverter.Convert(value);
528

```

```

519     [MethodImpl(MethodImplOptions.AggressiveInlining)]
520     protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
    ↪     second);
521
522     [MethodImpl(MethodImplOptions.AggressiveInlining)]
523     protected virtual TLink Subtract(TLink first, TLink second) =>
    ↪     Arithmetic<TLink>.Subtract(first, second);
524
525     [MethodImpl(MethodImplOptions.AggressiveInlining)]
526     protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
527
528     [MethodImpl(MethodImplOptions.AggressiveInlining)]
529     protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
530
531     #region Disposable
532
533     protected override bool AllowMultipleDisposeCalls
534     {
535         [MethodImpl(MethodImplOptions.AggressiveInlining)]
536         get => true;
537     }
538
539     [MethodImpl(MethodImplOptions.AggressiveInlining)]
540     protected override void Dispose(bool manual, bool wasDisposed)
541     {
542         if (!wasDisposed)
543         {
544             ResetPointers();
545             _dataMemory.DisposeIfPossible();
546             _indexMemory.DisposeIfPossible();
547         }
548     }
549
550     #endregion
551 }
552 }

```

## 1.95 ./csharp/Platform.Data.Doublets.SplitMemory.Generic/UnusedLinksListMethods.cs

```

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

```

```

37     protected override TLink GetPrevious(TLink element) =>
38         ↪ GetLinkDataPartReference(element).Source;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override TLink GetNext(TLink element) =>
42         ↪ GetLinkDataPartReference(element).Target;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override TLink GetSize() => GetHeaderReference().FreeLinks;
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
49         ↪ element;
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override void SetPrevious(TLink element, TLink previous) =>
53         ↪ GetLinkDataPartReference(element).Source = previous;
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void SetNext(TLink element, TLink next) =>
57         ↪ GetLinkDataPartReference(element).Target = next;
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
61 }

```

#### 1.96 ./csharp/Platform.Data.Doublets.SplitMemory/ILinksListMethods.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.SplitMemory
6  {
7      public interface ILinksListMethods<TLink>
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         void Detach(TLink freeLink);
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         void AttachAsFirst(TLink link);
14     }
15 }

```

#### 1.97 ./csharp/Platform.Data.Doublets.SplitMemory/ILinksTreeMethods.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.SplitMemory
8  {
9      public interface ILinksTreeMethods<TLink>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         TLink CountUsages(TLink root);
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         TLink Search(TLink source, TLink target);
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         TLink EachUsage(TLink root, Func<IList<TLink>, TLink> handler);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         void Detach(ref TLink root, TLink linkIndex);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         void Attach(ref TLink root, TLink linkIndex);
25     }
26 }

```

#### 1.98 ./csharp/Platform.Data.Doublets.SplitMemory/LinksHeaderIndexPart.cs

```

1  using System;
2  using System.Collections.Generic;

```

```

3 using System.Runtime.CompilerServices;
4 using Platform.Unsafe;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.SplitMemory
9 {
10     public struct LinksHeaderIndexPart<TLink> : IEquatable<LinksHeaderIndexPart<TLink>>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14
15         public static readonly long SizeInBytes = Structure<LinksHeaderIndexPart<TLink>>.Size;
16
17         public TLink AllocatedLinks;
18         public TLink ReservedLinks;
19         public TLink FreeLinks;
20         public TLink FirstFreeLink;
21         public TLink LastFreeLink;
22         public TLink Reserved6;
23         public TLink Reserved7;
24         public TLink Reserved8;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public override bool Equals(object obj) => obj is LinksHeaderIndexPart<TLink>
28             ↪ linksHeader ? Equals(linksHeader) : false;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool Equals(LinksHeaderIndexPart<TLink> other)
32             => _equalityComparer.Equals(AllocatedLinks, other.AllocatedLinks)
33             && _equalityComparer.Equals(ReservedLinks, other.ReservedLinks)
34             && _equalityComparer.Equals(FreeLinks, other.FreeLinks)
35             && _equalityComparer.Equals(FirstFreeLink, other.FirstFreeLink)
36             && _equalityComparer.Equals(LastFreeLink, other.LastFreeLink)
37             && _equalityComparer.Equals(Reserved6, other.Reserved6)
38             && _equalityComparer.Equals(Reserved7, other.Reserved7)
39             && _equalityComparer.Equals(Reserved8, other.Reserved8);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
43             ↪ FirstFreeLink, LastFreeLink, Reserved6, Reserved7, Reserved8).GetHashCode();
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public static bool operator ==(LinksHeaderIndexPart<TLink> left,
47             ↪ LinksHeaderIndexPart<TLink> right) => left.Equals(right);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static bool operator !=(LinksHeaderIndexPart<TLink> left,
51             ↪ LinksHeaderIndexPart<TLink> right) => !(left == right);
52     }
53 }

```

#### 1.99 ./csharp/Platform.Data.Doublets.SplitMemory/RawLinkDataPart.cs

```

1 using Platform.Unsafe;
2 using System;
3 using System.Collections.Generic;
4 using System.Runtime.CompilerServices;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.SplitMemory
9 {
10     public struct RawLinkDataPart<TLink> : IEquatable<RawLinkDataPart<TLink>>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14
15         public static readonly long SizeInBytes = Structure<RawLinkDataPart<TLink>>.Size;
16
17         public TLink Source;
18         public TLink Target;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public override bool Equals(object obj) => obj is RawLinkDataPart<TLink> link ?
22             ↪ Equals(link) : false;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public bool Equals(RawLinkDataPart<TLink> other)
26             => _equalityComparer.Equals(Source, other.Source)
27             && _equalityComparer.Equals(Target, other.Target);
28     }
29 }

```

```

27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     public override int GetHashCode() => (Source, Target).GetHashCode();
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     public static bool operator ==(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
32     ↪ right) => left.Equals(right);
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     public static bool operator !=(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
36     ↪ right) => !(left == right);
37 }
38 }

```

#### 1.100 ./csharp/Platform.Data.Doublets/SplitMemory/RawLinkIndexPart.cs

```

1  using Platform.Unsafe;
2  using System;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.SplitMemory
9  {
10     public struct RawLinkIndexPart<TLink> : IEquatable<RawLinkIndexPart<TLink>>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13         ↪ EqualityComparer<TLink>.Default;
14
15         public static readonly long SizeInBytes = Structure<RawLinkIndexPart<TLink>>.Size;
16
17         public TLink RootAsSource;
18         public TLink LeftAsSource;
19         public TLink RightAsSource;
20         public TLink SizeAsSource;
21         public TLink RootAsTarget;
22         public TLink LeftAsTarget;
23         public TLink RightAsTarget;
24         public TLink SizeAsTarget;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public override bool Equals(object obj) => obj is RawLinkIndexPart<TLink> link ?
28         ↪ Equals(link) : false;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool Equals(RawLinkIndexPart<TLink> other)
32         => _equalityComparer.Equals(RootAsSource, other.RootAsSource)
33         && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
34         && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
35         && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
36         && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
37         && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
38         && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
39         && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public override int GetHashCode() => (RootAsSource, LeftAsSource, RightAsSource,
43         ↪ SizeAsSource, RootAsTarget, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public static bool operator ==(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
47         ↪ right) => left.Equals(right);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static bool operator !=(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
51         ↪ right) => !(left == right);
52     }
53 }

```

#### 1.101 ./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 =
12         ↳ EqualityComparer<TLink>.Default;
13
14     private readonly TLink _stack;
15
16     public bool IsEmpty
17     {
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         get => _equalityComparer.Equals(Peek(), _stack);
20     }
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     public Stack(ILinks<TLink> links, TLink stack) : base(links) => _stack = stack;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     private TLink GetStackMarker() => _links.GetSource(_stack);
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     private TLink GetTop() => _links.GetTarget(_stack);
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     public TLink Peek() => _links.GetTarget(GetTop());
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     public TLink Pop()
36     {
37         var element = Peek();
38         if (!_equalityComparer.Equals(element, _stack))
39         {
40             var top = GetTop();
41             var previousTop = _links.GetSource(top);
42             _links.Update(_stack, GetStackMarker(), previousTop);
43             _links.Delete(top);
44         }
45         return element;
46     }
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
50         ↳ _links.GetOrCreate(GetTop(), element));

```

#### 1.102 ./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Stacks
6 {
7     public static class StackExtensions
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
11        {
12            var stackPoint = links.CreatePoint();
13            var stack = links.Update(stackPoint, stackMarker, stackPoint);
14            return stack;
15        }
16    }
17 }

```

#### 1.103 ./csharp/Platform.Data.Doublets/SynchronizedLinks.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Data.Doublets;
5 using Platform.Threading.Synchronization;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets
10 {
11     /// <remarks>
12     /// TODO: Autogeneration of synchronized wrapper (decorator).
13     /// TODO: Try to unfold code of each method using IL generation for performance improvements.
14     /// TODO: Or even to unfold multiple layers of implementations.
15     /// </remarks>
16     public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>

```

```

17 {
18     public LinksConstants<TLinkAddress> Constants
19     {
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         get;
22     }
23
24     public ISynchronization SyncRoot
25     {
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         get;
28     }
29
30     public ILinks<TLinkAddress> Sync
31     {
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         get;
34     }
35
36     public ILinks<TLinkAddress> Unsync
37     {
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         get;
40     }
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
44         ↳ ReaderWriterLockSynchronization(), links) { }
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
48     {
49         SyncRoot = synchronization;
50         Sync = this;
51         Unsync = links;
52         Constants = links.Constants;
53     }
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public TLinkAddress Count(IList<TLinkAddress> restriction) =>
57         ↳ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
61         ↳ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
62         ↳ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
66         ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
70         ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
71         ↳ Unsync.Update);
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public void Delete(IList<TLinkAddress> restrictions) =>
75         ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
76
77     //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
78     ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
79     //{
80     //    if (restriction != null && substitution != null &&
81     ↳ !substitution.EqualTo(restriction))
82     //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
83     ↳ substitution, substitutedHandler, Unsync.Trigger);
84
85     //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
86     ↳ substitutedHandler, Unsync.Trigger);
87     //}
88 }
89 }

```

#### 1.104 ./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs

```

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

```

```

5 using Platform.Singletons;
6 using Platform.Data.Doublets.Unicode;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets
11 {
12     public static class UInt64LinksExtensions
13     {
14         public static readonly LinksConstants<ulong> Constants =
15             ↳ Default<LinksConstants<ulong>>.Instance;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
22         {
23             if (sequence == null)
24             {
25                 return false;
26             }
27             var constants = links.Constants;
28             for (var i = 0; i < sequence.Length; i++)
29             {
30                 if (sequence[i] == constants.Any)
31                 {
32                     return true;
33                 }
34             }
35             return false;
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
40             ↳ Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
41             ↳ false)
42         {
43             var sb = new StringBuilder();
44             var visited = new HashSet<ulong>();
45             links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
46                 ↳ innerSb.Append(link.Index), renderIndex, renderDebug);
47             return sb.ToString();
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
52             ↳ Func<Link<ulong>, bool> isElement, Action<StringBuilder, Link<ulong>> appendElement,
53             ↳ bool renderIndex = false, bool renderDebug = false)
54         {
55             var sb = new StringBuilder();
56             var visited = new HashSet<ulong>();
57             links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
58                 ↳ renderDebug);
59             return sb.ToString();
60         }
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
64             ↳ HashSet<ulong> visited, ulong linkIndex, Func<Link<ulong>, bool> isElement,
65             ↳ Action<StringBuilder, Link<ulong>> appendElement, bool renderIndex = false, bool
66             ↳ renderDebug = false)
67         {
68             if (sb == null)
69             {
70                 throw new ArgumentNullException(nameof(sb));
71             }
72             if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
73                 ↳ Constants.Itself)
74             {
75                 return;
76             }
77             if (links.Exists(linkIndex))
78             {
79                 if (visited.Add(linkIndex))
80                 {
81                     sb.Append('(');
82                     var link = new Link<ulong>(links.GetLink(linkIndex));
83                     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,
91                                     ↪ appendElement, renderIndex);
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,
108                                         ↪ appendElement, renderIndex);
109                 }
110             }
111             sb.Append('');
112         }
113         else
114         {
115             if (renderDebug)
116             {
117                 sb.Append('*');
118             }
119             sb.Append(linkIndex);
120         }
121     }
122     else
123     {
124         if (renderDebug)
125         {
126             sb.Append('~');
127         }
128         sb.Append(linkIndex);
129     }
130 }
131 }

```

#### 1.105 ./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;
12 using Platform.Data.Doublets.Decorators;
13 using Platform.Exceptions;
14
15 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16

```

```

17 namespace Platform.Data.Doublets
18 {
19     public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
20     {
21         /// <remarks>
22         /// Альтернативные варианты хранения трансформации (элемента транзакции):
23         ///
24         /// private enum TransitionType
25         /// {
26         ///     Creation,
27         ///     UpdateOf,
28         ///     UpdateTo,
29         ///     Deletion
30         /// }
31         ///
32         /// private struct Transition
33         /// {
34         ///     public ulong TransactionId;
35         ///     public UniqueTimestamp Timestamp;
36         ///     public TransactionItemType Type;
37         ///     public Link Source;
38         ///     public Link Linker;
39         ///     public Link Target;
40         /// }
41         /// Или
42         ///
43         /// public struct TransitionHeader
44         /// {
45         ///     public ulong TransactionIdCombined;
46         ///     public ulong TimestampCombined;
47         ///
48         ///     public ulong TransactionId
49         ///     {
50         ///         get
51         ///         {
52         ///             return (ulong) mask & TransactionIdCombined;
53         ///         }
54         ///     }
55         ///
56         ///     public UniqueTimestamp Timestamp
57         ///     {
58         ///         get
59         ///         {
60         ///             return (UniqueTimestamp)mask & TransactionIdCombined;
61         ///         }
62         ///     }
63         ///
64         ///     public TransactionItemType Type
65         ///     {
66         ///         get
67         ///         {
68         ///             // Использовать по одному биту из TransactionId и Timestamp,
69         ///             // для значения в 2 бита, которое представляет тип операции
70         ///             throw new NotImplementedException();
71         ///         }
72         ///     }
73         /// }
74         /// }
75         ///
76         /// private struct Transition
77         /// {
78         ///     public TransitionHeader Header;
79         ///     public Link Source;
80         ///     public Link Linker;
81         ///     public Link Target;
82         /// }
83         ///
84         /// </remarks>
85         public struct Transition : IEquatable<Transition>
86         {
87             public static readonly long Size = Structure<Transition>.Size;
88
89             public readonly ulong TransactionId;
90             public readonly Link<ulong> Before;
91             public readonly Link<ulong> After;
92             public readonly Timestamp Timestamp;
93
94             [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

95     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
    ↪ transactionId, Link<ulong> before, Link<ulong> after)
96     {
97         TransactionId = transactionId;
98         Before = before;
99         After = after;
100         Timestamp = uniqueTimestampFactory.Create();
101     }
102
103     [MethodImpl(MethodImplOptions.AggressiveInlining)]
104     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
    ↪ transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
    ↪ before, default) { }
105
106     [MethodImpl(MethodImplOptions.AggressiveInlining)]
107     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
    ↪ transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
    ↪ }
108
109     [MethodImpl(MethodImplOptions.AggressiveInlining)]
110     public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
    ↪ {After}";
111
112     [MethodImpl(MethodImplOptions.AggressiveInlining)]
113     public override bool Equals(object obj) => obj is Transition transition ?
    ↪ Equals(transition) : false;
114
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     public override int GetHashCode() => (TransactionId, Before, After,
    ↪ Timestamp).GetHashCode();
117
118     [MethodImpl(MethodImplOptions.AggressiveInlining)]
119     public bool Equals(Transition other) => TransactionId == other.TransactionId &&
    ↪ Before == other.Before && After == other.After && Timestamp == other.Timestamp;
120
121     [MethodImpl(MethodImplOptions.AggressiveInlining)]
122     public static bool operator ==(Transition left, Transition right) =>
    ↪ left.Equals(right);
123
124     [MethodImpl(MethodImplOptions.AggressiveInlining)]
125     public static bool operator !=(Transition left, Transition right) => !(left ==
    ↪ right);
126 }
127
128 /// <remarks>
129 /// Другие варианты реализации транзакций (атомарности):
130 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
    ↪ Target)) и индексов.
131 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
    ↪ потребуется решить вопрос
132 /// со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
    ↪ пересечениями идентификаторов.
133 ///
134 /// Где хранить промежуточный список транзакций?
135 ///
136 /// В оперативной памяти:
137 /// Минусы:
138 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
139 /// так как нужно отдельно выделять память под список трансформаций.
140 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
141 /// если транзакция использует слишком много трансформаций.
142 /// -> Можно использовать жёсткий диск для слишком длинных транзакций.
143 /// -> Максимальный размер списка трансформаций можно ограничить / задать
    ↪ константой.
144 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
    ↪ создавая задержку.
145 ///
146 /// На жёстком диске:
147 /// Минусы:
148 /// 1. Длительный отклик, на запись каждой трансформации.
149 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
150 /// -> Это может решаться упаковкой/исключением дублирующих операций.
151 /// -> Также это может решаться тем, что короткие транзакции вообще
152 /// не будут записываться в случае отката.
153 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
    ↪ операции (трансформации)
154 /// будут записаны в лог.

```

```

155 ///
156 /// </remarks>
157 public class Transaction : DisposableBase
158 {
159     private readonly Queue<Transition> _transitions;
160     private readonly UInt64LinksTransactionsLayer _layer;
161     public bool IsCommitted { get; private set; }
162     public bool IsReverted { get; private set; }
163
164     [MethodImpl(MethodImplOptions.AggressiveInlining)]
165     public Transaction(UInt64LinksTransactionsLayer layer)
166     {
167         _layer = layer;
168         if (_layer._currentTransactionId != 0)
169         {
170             throw new NotSupportedException("Nested transactions not supported.");
171         }
172         IsCommitted = false;
173         IsReverted = false;
174         _transitions = new Queue<Transition>();
175         SetCurrentTransaction(layer, this);
176     }
177
178     [MethodImpl(MethodImplOptions.AggressiveInlining)]
179     public void Commit()
180     {
181         EnsureTransactionAllowsWriteOperations(this);
182         while (_transitions.Count > 0)
183         {
184             var transition = _transitions.Dequeue();
185             _layer._transitions.Enqueue(transition);
186         }
187         _layer._lastCommittedTransactionId = _layer._currentTransactionId;
188         IsCommitted = true;
189     }
190
191     [MethodImpl(MethodImplOptions.AggressiveInlining)]
192     private void Revert()
193     {
194         EnsureTransactionAllowsWriteOperations(this);
195         var transitionsToRevert = new Transition[_transitions.Count];
196         _transitions.CopyTo(transitionsToRevert, 0);
197         for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
198         {
199             _layer.RevertTransition(transitionsToRevert[i]);
200         }
201         IsReverted = true;
202     }
203
204     [MethodImpl(MethodImplOptions.AggressiveInlining)]
205     public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
206 ↪ Transaction transaction)
207     {
208         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
209         layer._currentTransactionTransitions = transaction._transitions;
210         layer._currentTransaction = transaction;
211     }
212
213     [MethodImpl(MethodImplOptions.AggressiveInlining)]
214     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
215     {
216         if (transaction.IsReverted)
217         {
218             throw new InvalidOperationException("Transation is reverted.");
219         }
220         if (transaction.IsCommitted)
221         {
222             throw new InvalidOperationException("Transation is committed.");
223         }
224     }
225
226     [MethodImpl(MethodImplOptions.AggressiveInlining)]
227     protected override void Dispose(bool manual, bool wasDisposed)
228     {
229         if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
230         {
231             if (!IsCommitted && !IsReverted)
232             {
233                 Revert();

```

```

233         }
234         _layer.ResetCurrentTransation();
235     }
236 }
237
238
239 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
240
241 private readonly string _logAddress;
242 private readonly FileStream _log;
243 private readonly Queue<Transition> _transitions;
244 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
245 private Task _transitionsPusher;
246 private Transition _lastCommittedTransition;
247 private ulong _currentTransactionId;
248 private Queue<Transition> _currentTransactionTransitions;
249 private Transaction _currentTransaction;
250 private ulong _lastCommittedTransactionId;
251
252 [MethodImpl(MethodImplOptions.AggressiveInlining)]
253 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
254     : base(links)
255 {
256     if (string.IsNullOrEmpty(logAddress))
257     {
258         throw new ArgumentNullException(nameof(logAddress));
259     }
260     // В первой строке файла хранится последняя закоммиченную транзакцию.
261     // При запуске это используется для проверки удачного закрытия файла лога.
262     // In the first line of the file the last committed transaction is stored.
263     // On startup, this is used to check that the log file is successfully closed.
264     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
265     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
266     if (!lastCommittedTransition.Equals(lastWrittenTransition))
267     {
268         Dispose();
269         throw new NotSupportedException("Database is damaged, autorecovery is not
270             ↳ supported yet.");
271     }
272     if (lastCommittedTransition == default)
273     {
274         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
275     }
276     _lastCommittedTransition = lastCommittedTransition;
277     // TODO: Think about a better way to calculate or store this value
278     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
279     _lastCommittedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
280         ↳ x.TransactionId) : 0;
281     _uniqueTimestampFactory = new UniqueTimestampFactory();
282     _logAddress = logAddress;
283     _log = FileHelpers.Append(logAddress);
284     _transitions = new Queue<Transition>();
285     _transitionsPusher = new Task(TransitionsPusher);
286     _transitionsPusher.Start();
287 }
288
289 [MethodImpl(MethodImplOptions.AggressiveInlining)]
290 public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);
291
292 [MethodImpl(MethodImplOptions.AggressiveInlining)]
293 public override ulong Create(IList<ulong> restrictions)
294 {
295     var createdLinkIndex = _links.Create();
296     var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
297     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
298         ↳ default, createdLink));
299     return createdLinkIndex;
300 }
301
302 [MethodImpl(MethodImplOptions.AggressiveInlining)]
303 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
304 {
305     var linkIndex = restrictions[_constants.IndexPart];
306     var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
307     linkIndex = _links.Update(restrictions, substitution);
308     var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
309     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
310         ↳ beforeLink, afterLink));
311     return linkIndex;
312 }

```

```

308     }
309
310     [MethodImpl(MethodImplOptions.AggressiveInlining)]
311     public override void Delete(ICollection<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,
317             ↪ deletedLink, default));
318     }
319
320     [MethodImpl(MethodImplOptions.AggressiveInlining)]
321     private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
322         ↪ _transitions;
323
324     [MethodImpl(MethodImplOptions.AggressiveInlining)]
325     private void CommitTransition(Transition transition)
326     {
327         if (_currentTransaction != null)
328         {
329             Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
330         }
331         var transitions = GetCurrentTransitions();
332         transitions.Enqueue(transition);
333     }
334
335     [MethodImpl(MethodImplOptions.AggressiveInlining)]
336     private void RevertTransition(Transition transition)
337     {
338         if (transition.After.IsNull()) // Revert Deletion with Creation
339         {
340             _links.Create();
341         }
342         else if (transition.Before.IsNull()) // Revert Creation with Deletion
343         {
344             _links.Delete(transition.After.Index);
345         }
346         else // Revert Update
347         {
348             _links.Update(new[] { transition.After.Index, transition.Before.Source,
349                 ↪ transition.Before.Target });
350         }
351     }
352
353     [MethodImpl(MethodImplOptions.AggressiveInlining)]
354     private void ResetCurrentTransation()
355     {
356         _currentTransactionId = 0;
357         _currentTransactionTransitions = null;
358         _currentTransaction = null;
359     }
360
361     [MethodImpl(MethodImplOptions.AggressiveInlining)]
362     private void PushTransitions()
363     {
364         if (_log == null || _transitions == null)
365         {
366             return;
367         }
368         for (var i = 0; i < _transitions.Count; i++)
369         {
370             var transition = _transitions.Dequeue();
371
372             _log.Write(transition);
373             _lastCommittedTransition = transition;
374         }
375     }
376
377     [MethodImpl(MethodImplOptions.AggressiveInlining)]
378     private void TransitionsPusher()
379     {
380         while (!Disposable.IsDisposed && _transitionsPusher != null)
381         {
382             Thread.Sleep(DefaultPushDelay);
383             PushTransitions();
384         }
385     }

```

```

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, _lastCommittedTransition);
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 }

```

#### 1.106 ./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>,
9     ↪ IConverter<char, TLink>
10     {
11         private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
12         ↪ UncheckedConverter<char, TLink>.Default;
13
14         private readonly IConverter<TLink> _addressToNumberConverter;
15         private readonly TLink _unicodeSymbolMarker;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
19         ↪ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
20         {
21             _addressToNumberConverter = addressToNumberConverter;
22             _unicodeSymbolMarker = unicodeSymbolMarker;
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public TLink Convert(char source)
27         {
28             var unaryNumber =
29             ↪ _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
30             return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
31         }
32     }
33 }

```

### 1.107 ./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs

```

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<char, TLink> _charToUnicodeSymbolConverter;
13         private readonly ISequenceIndex<TLink> _index;
14         private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
15         private readonly TLink _unicodeSequenceMarker;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
        ↳ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
        ↳ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
19         {
20             _charToUnicodeSymbolConverter = charToUnicodeSymbolConverter;
21             _index = index;
22             _listToSequenceLinkConverter = listToSequenceLinkConverter;
23             _unicodeSequenceMarker = unicodeSequenceMarker;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(string source)
28         {
29             var elements = new TLink[source.Length];
30             for (int i = 0; i < elements.Length; i++)
31             {
32                 elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
33             }
34             _index.Add(elements);
35             var sequence = _listToSequenceLinkConverter.Convert(elements);
36             return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
37         }
38     }
39 }

```

### 1.108 ./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using Platform.Data.Sequences;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeMap
13     {
14         public static readonly ulong FirstCharLink = 1;
15         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
16         public static readonly ulong MapSize = 1 + char.MaxValue;
17
18         private readonly ILinks<ulong> _links;
19         private bool _initialized;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public UnicodeMap(ILinks<ulong> links) => _links = links;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public static UnicodeMap InitNew(ILinks<ulong> links)
26         {
27             var map = new UnicodeMap(links);
28             map.Init();
29             return map;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public void Init()
34         {
35             if (!_initialized)
36             {

```



```

37         return;
38     }
39     _initialized = true;
40     var firstLink = _links.CreatePoint();
41     if (firstLink != FirstCharLink)
42     {
43         _links.Delete(firstLink);
44     }
45     else
46     {
47         for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
48         {
49             // From NIL to It (NIL -> Character) transformation meaning, (or infinite
50             ↪ amount of NIL characters before actual Character)
51             var createdLink = _links.CreatePoint();
52             _links.Update(createdLink, firstLink, createdLink);
53             if (createdLink != i)
54             {
55                 throw new InvalidOperationException("Unable to initialize UTF 16
56                 ↪ table.");
57             }
58         }
59     }
60     // 0 - null link
61     // 1 - nil character (0 character)
62     // ...
63     // 65536 (0(1) + 65535 = 65536 possible values)
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     public static ulong FromCharToLink(char character) => (ulong)character + 1;
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     public static char FromLinkToChar(ulong link) => (char)(link - 1);
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     public static bool IsCharLink(ulong link) => link <= MapSize;
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     public static string FromLinksToString(IList<ulong> linksList)
76     {
77         var sb = new StringBuilder();
78         for (int i = 0; i < linksList.Count; i++)
79         {
80             sb.Append(FromLinkToChar(linksList[i]));
81         }
82         return sb.ToString();
83     }
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
87     {
88         var sb = new StringBuilder();
89         if (links.Exists(link))
90         {
91             StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
92             ↪ x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
93             ↪ element =>
94             {
95                 sb.Append(FromLinkToChar(element));
96                 return true;
97             });
98         }
99         return sb.ToString();
100     }
101
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
104     ↪ chars.Length);
105
106     [MethodImpl(MethodImplOptions.AggressiveInlining)]
107     public static ulong[] FromCharsToLinkArray(char[] chars, int count)
108     {
109         // char array to ulong array
110         var linksSequence = new ulong[count];
111         for (var i = 0; i < count; i++)
112         {

```

```

111         linksSequence[i] = FromCharToLink(chars[i]);
112     }
113     return linksSequence;
114 }
115
116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 public static ulong[] FromStringToLinkArray(string sequence)
118 {
119     // char array to ulong array
120     var linksSequence = new ulong[sequence.Length];
121     for (var i = 0; i < sequence.Length; i++)
122     {
123         linksSequence[i] = FromCharToLink(sequence[i]);
124     }
125     return linksSequence;
126 }
127
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
130 {
131     var result = new List<ulong[]>();
132     var offset = 0;
133     while (offset < sequence.Length)
134     {
135         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
136         var relativeLength = 1;
137         var absoluteLength = offset + relativeLength;
138         while (absoluteLength < sequence.Length &&
139             currentCategory ==
140                 CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
141         {
142             relativeLength++;
143             absoluteLength++;
144         }
145         // char array to ulong array
146         var innerSequence = new ulong[relativeLength];
147         var maxLength = offset + relativeLength;
148         for (var i = offset; i < maxLength; i++)
149         {
150             innerSequence[i - offset] = FromCharToLink(sequence[i]);
151         }
152         result.Add(innerSequence);
153         offset += relativeLength;
154     }
155     return result;
156 }
157
158 [MethodImpl(MethodImplOptions.AggressiveInlining)]
159 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
160 {
161     var result = new List<ulong[]>();
162     var offset = 0;
163     while (offset < array.Length)
164     {
165         var relativeLength = 1;
166         if (array[offset] <= LastCharLink)
167         {
168             var currentCategory =
169                 CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
170             var absoluteLength = offset + relativeLength;
171             while (absoluteLength < array.Length &&
172                 array[absoluteLength] <= LastCharLink &&
173                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
174                     array[absoluteLength])))
175             {
176                 relativeLength++;
177                 absoluteLength++;
178             }
179         }
180         else
181         {
182             var absoluteLength = offset + relativeLength;
183             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
184             {
185                 relativeLength++;
186                 absoluteLength++;
187             }
188         }
189         // copy array

```

```

187         var innerSequence = new ulong[relativeLength];
188         var maxLength = offset + relativeLength;
189         for (var i = offset; i < maxLength; i++)
190         {
191             innerSequence[i - offset] = array[i];
192         }
193         result.Add(innerSequence);
194         offset += relativeLength;
195     }
196     return result;
197 }
198 }
199 }

```

#### 1.109 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.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.Unicode
8 {
9     public class UnicodeSequenceCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
10         ↳ ICriterionMatcher<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↳ EqualityComparer<TLink>.Default;
14
15         private readonly TLink _unicodeSequenceMarker;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public UnicodeSequenceCriterionMatcher(ILinks<TLink> links, TLink unicodeSequenceMarker)
19             ↳ : base(links) => _unicodeSequenceMarker = unicodeSequenceMarker;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public bool IsMatched(TLink link) => _equalityComparer.Equals(_links.GetTarget(link),
23             ↳ _unicodeSequenceMarker);
24     }
25 }

```

#### 1.110 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs

```

1 using System;
2 using System.Linq;
3 using System.Runtime.CompilerServices;
4 using Platform.Interfaces;
5 using Platform.Converters;
6 using Platform.Data.Doublets.Sequences.Walkers;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
13         ↳ IConverter<TLink, string>
14     {
15         private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
16         private readonly ISequenceWalker<TLink> _sequenceWalker;
17         private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
21             ↳ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
22             ↳ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
23         {
24             _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
25             _sequenceWalker = sequenceWalker;
26             _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public string Convert(TLink source)
31         {
32             if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
33             {
34                 throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
35                     ↳ not a unicode sequence.");
36             }
37             var sequence = _links.GetSource(source);
38         }
39     }
40 }

```

```

34         var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
35             ↪ Convert).ToArray();
36         return new string(charArray);
37     }
38 }

```

#### 1.111 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.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.Unicode
8 {
9     public class UnicodeSymbolCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
10         ↪ ICriterionMatcher<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14
15         private readonly TLink _unicodeSymbolMarker;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public UnicodeSymbolCriterionMatcher(ILinks<TLink> links, TLink unicodeSymbolMarker) :
19             ↪ base(links) => _unicodeSymbolMarker = unicodeSymbolMarker;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public bool IsMatched(TLink link) => _equalityComparer.Equals(_links.GetTarget(link),
23             ↪ _unicodeSymbolMarker);
24     }
25 }

```

#### 1.112 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs

```

1 using System;
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.Unicode
9 {
10     public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<TLink, char>
12     {
13         private static readonly UncheckedConverter<TLink, char> _addressToCharConverter =
14             ↪ UncheckedConverter<TLink, char>.Default;
15
16         private readonly IConverter<TLink> _numberToAddressConverter;
17         private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
21             ↪ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
22             ↪ base(links)
23         {
24             _numberToAddressConverter = numberToAddressConverter;
25             _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
26         }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public char Convert(TLink source)
30         {
31             if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
32             {
33                 throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
34                     ↪ not a unicode symbol.");
35             }
36             return _addressToCharConverter.Convert(_numberToAddressConverter.Convert(_links.GetS
37                 ↪ ource(source)));
38         }
39     }
40 }

```

#### 1.113 ./csharp/Platform.Data.Doublets.Tests/ComparisonTests.cs

```

1 using System;
2 using System.Collections.Generic;

```

```

3 using Xunit;
4 using Platform.Diagnostics;
5
6 namespace Platform.Data.Doublets.Tests
7 {
8     public static class ComparisonTests
9     {
10         private class UInt64Comparer : IComparer

```

#### 1.114 ./csharp/Platform.Data.Doublets.Tests/EqualityTests.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Xunit;
4 using Platform.Diagnostics;
5
6 namespace Platform.Data.Doublets.Tests
7 {
8     public static class EqualityTests
9     {
10         protected class UInt64EqualityComparer : IEqualityComparer<ulong>
11         {
12             public bool Equals(ulong x, ulong y) => x == y;
13         }
14     }
15 }

```

```

13     public int GetHashCode(ulong obj) => obj.GetHashCode();
14 }
15
16 private static bool Equals1<T>(T x, T y) => Equals(x, y);
17
18 private static bool Equals2<T>(T x, T y) => x.Equals(y);
19
20 private static bool Equals3(ulong x, ulong y) => x == y;
21
22 [Fact]
23 public static void EqualsPerformanceTest()
24 {
25     const int N = 1000000;
26
27     ulong x = 10;
28     ulong y = 500;
29
30     bool result = false;
31
32     var ts1 = Performance.Measure(() =>
33     {
34         for (int i = 0; i < N; i++)
35         {
36             result = Equals1(x, y);
37         }
38     });
39
40     var ts2 = Performance.Measure(() =>
41     {
42         for (int i = 0; i < N; i++)
43         {
44             result = Equals2(x, y);
45         }
46     });
47
48     var ts3 = Performance.Measure(() =>
49     {
50         for (int i = 0; i < N; i++)
51         {
52             result = Equals3(x, y);
53         }
54     });
55
56     var equalityComparer1 = EqualityComparer<ulong>.Default;
57
58     var ts4 = Performance.Measure(() =>
59     {
60         for (int i = 0; i < N; i++)
61         {
62             result = equalityComparer1.Equals(x, y);
63         }
64     });
65
66     var equalityComparer2 = new UInt64EqualityComparer();
67
68     var ts5 = Performance.Measure(() =>
69     {
70         for (int i = 0; i < N; i++)
71         {
72             result = equalityComparer2.Equals(x, y);
73         }
74     });
75
76     Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;
77
78     var ts6 = Performance.Measure(() =>
79     {
80         for (int i = 0; i < N; i++)
81         {
82             result = equalityComparer3(x, y);
83         }
84     });
85
86     var comparer = Comparer<ulong>.Default;
87
88     var ts7 = Performance.Measure(() =>
89     {
90         for (int i = 0; i < N; i++)
91         {
92

```

```

93         result = comparer.Compare(x, y) == 0;
94     }
95 });
96
97     Assert.True(ts2 < ts1);
98     Assert.True(ts3 < ts2);
99     Assert.True(ts5 < ts4);
100    Assert.True(ts5 < ts6);
101
102    Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");
103 }
104 }
105 }

```

#### 1.115 ./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.ResizableDirectMemory.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
34                 ↳ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
35                 ↳ implementation of tree cuts out 5 bits from the address space.
36             Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
37                 ↳ stMultipleRandomCreationsAndDeletions(100));
38             Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
39                 ↳ MultipleRandomCreationsAndDeletions(100));
40             Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
41                 ↳ tMultipleRandomCreationsAndDeletions(100));
42         }
43
44         private static void Using<TLink>(Action<ILinks<TLink>> action)
45         {
46             using (var scope = new Scope<Types<HeapResizableDirectMemory,
47                 ↳ ResizableDirectMemoryLinks<TLink>>>())
48             {
49                 action(scope.Use<ILinks<TLink>>());
50             }
51         }
52     }
53 }

```

#### 1.116 ./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs

```

1  using Xunit;
2
3  namespace Platform.Data.Doublets.Tests
4  {
5     public static class LinksConstantsTests
6     {
7         [Fact]
8         public static void ExternalReferencesTest()

```

```

9      {
10         LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
            ↳ (long.MaxValue + 1UL, ulong.MaxValue));
11
12         //var minimum = new Hybrid<ulong>(0, isExternal: true);
13         var minimum = new Hybrid<ulong>(1, isExternal: true);
14         var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);
15
16         Assert.True(constants.IsExternalReference(minimum));
17         Assert.True(constants.IsExternalReference(maximum));
18     }
19 }
20 }

```

### 1.117 ./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs

```

1  using System;
2  using System.Linq;
3  using Xunit;
4  using Platform.Collections.Stacks;
5  using Platform.Collections.Arrays;
6  using Platform.Memory;
7  using Platform.Data.Numbers.Raw;
8  using Platform.Data.Doublets.Sequences;
9  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
10 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
11 using Platform.Data.Doublets.Sequences.Converters;
12 using Platform.Data.Doublets.PropertyOperators;
13 using Platform.Data.Doublets.Incrementers;
14 using Platform.Data.Doublets.Sequences.Walkers;
15 using Platform.Data.Doublets.Sequences.Indexes;
16 using Platform.Data.Doublets.Unicode;
17 using Platform.Data.Doublets.Numbers.Unary;
18 using Platform.Data.Doublets.Decorators;
19 using Platform.Data.Doublets.ResizableDirectMemory.Specific;
20
21 namespace Platform.Data.Doublets.Tests
22 {
23     public static class OptimalVariantSequenceTests
24     {
25         private static readonly string _sequenceExample = "зеленела зелёная зелень";
26         private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet,
            ↳ consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
            ↳ magna aliqua.
27 Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
28 Et malesuada fames ac turpis egestas sed.
29 Eget velit aliquet sagittis id consectetur purus.
30 Dignissim cras tincidunt lobortis feugiat vivamus.
31 Vitae aliquet nec ullamcorper sit.
32 Lectus quam id leo in vitae.
33 Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
34 Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
35 Integer eget aliquet nibh praesent tristique.
36 Vitae congue eu consequat ac felis donec et odio.
37 Tristique et egestas quis ipsum suspendisse.
38 Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
39 Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
40 Imperdiet proin fermentum leo vel orci.
41 In ante metus dictum at tempor commodo.
42 Nisi lacus sed viverra tellus in.
43 Quam vulputate dignissim suspendisse in.
44 Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
45 Gravida cum sociis natoque penatibus et magnis dis parturient.
46 Risus quis varius quam quisque id diam.
47 Congue nisi vitae suscipit tellus mauris a diam maecenas.
48 Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
49 Pharetra vel turpis nunc eget lorem dolor sed viverra.
50 Mattis pellentesque id nibh tortor id aliquet.
51 Purus non enim praesent elementum facilisis leo vel.
52 Etiam sit amet nisl purus in mollis nunc sed.
53 Tortor at auctor urna nunc id cursus metus aliquam.
54 Volutpat odio facilisis mauris sit amet.
55 Turpis egestas pretium aenean pharetra magna ac placerat.
56 Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
57 Porttitor leo a diam sollicitudin tempor id eu.
58 Volutpat sed cras ornare arcu dui.
59 Ut aliquam purus sit amet luctus venenatis lectus magna.
60 Aliquet risus feugiat in ante metus dictum at.
61 Mattis nunc sed blandit libero.
62 Elit pellentesque habitant morbi tristique senectus et netus.
63 Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
64 Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
65 Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
66 Diam donec adipiscing tristique risus nec feugiat.

```



Pulvinar mattis nunc sed blandit libero volutpat.  
Cras fermentum odio eu feugiat pretium nibh ipsum.  
In nulla posuere sollicitudin aliquam ultrices sagittis orci a.  
Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.  
A iaculis at erat pellentesque.  
Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.  
Eget lorem dolor sed viverra ipsum nunc.  
Leo a diam sollicitudin tempor id eu.  
Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";

[Fact]

```
public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
{
    using (var scope = new TempLinksTestScope(useSequences: false))
    {
        var links = scope.Links;
        var constants = links.Constants;

        links.UseUnicode();

        var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

        var meaningRoot = links.CreatePoint();
        var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
        var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
        var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
            ↪ constants.Itself);

        var unaryNumberToAddressConverter = new
            ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
        var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
        var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
            ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
        var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
            ↪ frequencyPropertyMarker, frequencyMarker);
        var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
            ↪ frequencyPropertyOperator, frequencyIncrementer);
        var linkToItsFrequencyNumberConverter = new
            ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
            ↪ unaryNumberToAddressConverter);
        var sequenceToItsLocalElementLevelsConverter = new
            ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
            ↪ linkToItsFrequencyNumberConverter);
        var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
            ↪ sequenceToItsLocalElementLevelsConverter);

        var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
            ↪ Walker = new LeveledSequenceWalker<ulong>(links) });

        ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
            ↪ index, optimalVariantConverter);
    }
}
```

[Fact]

```
public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
{
    using (var scope = new TempLinksTestScope(useSequences: false))
    {
        var links = scope.Links;

        links.UseUnicode();

        var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

        var totalSequenceSymbolFrequencyCounter = new
            ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);

        var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
            ↪ totalSequenceSymbolFrequencyCounter);

        var index = new
            ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
        var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);

        var sequenceToItsLocalElementLevelsConverter = new
            ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
            ↪ linkToItsFrequencyNumberConverter);
```

```

128         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
129             ↪ sequenceToItsLocalElementLevelsConverter);
130
131         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
132             ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
133
134         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
135             ↪ index, optimalVariantConverter);
136     }
137
138     private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
139         ↪ SequenceToItsLocalElementLevelsConverter<ulong>
140         ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
141         ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
142     {
143         index.Add(sequence);
144
145         var optimalVariant = optimalVariantConverter.Convert(sequence);
146
147         var readSequence1 = sequences.ToList(optimalVariant);
148
149         Assert.True(sequence.SequenceEqual(readSequence1));
150     }
151
152     [Fact]
153     public static void SavedSequencesOptimizationTest()
154     {
155         LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
156             ↪ (long.MaxValue + 1UL, ulong.MaxValue));
157
158         using (var memory = new HeapResizableDirectMemory())
159         using (var disposableLinks = new UInt64ResizableDirectMemoryLinks(memory,
160             ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep, constants,
161             ↪ useAvlBasedIndex: false))
162         {
163             var links = new UInt64Links(disposableLinks);
164
165             var root = links.CreatePoint();
166
167             //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
168             var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
169
170             var unicodeSymbolMarker = links.GetOrCreate(root,
171                 ↪ addressToNumberConverter.Convert(1));
172             var unicodeSequenceMarker = links.GetOrCreate(root,
173                 ↪ addressToNumberConverter.Convert(2));
174
175             var totalSequenceSymbolFrequencyCounter = new
176                 ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
177             var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
178                 ↪ totalSequenceSymbolFrequencyCounter);
179             var index = new
180                 ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
181             var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
182                 ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
183             var sequenceToItsLocalElementLevelsConverter = new
184                 ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
185                 ↪ linkToItsFrequencyNumberConverter);
186             var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
187                 ↪ sequenceToItsLocalElementLevelsConverter);
188
189             var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
190                 ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
191
192             var unicodeSequencesOptions = new SequencesOptions<ulong>()
193             {
194                 UseSequenceMarker = true,
195                 SequenceMarkerLink = unicodeSequenceMarker,
196                 UseIndex = true,
197                 Index = index,
198                 LinksToSequenceConverter = optimalVariantConverter,
199                 Walker = walker,
200                 UseGarbageCollection = true
201             };
202
203             var unicodeSequences = new Sequences.Sequences(new
204                 ↪ SynchronizedLinks<ulong>(links), unicodeSequencesOptions);

```

```

186
187 // Create some sequences
188 var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
    ↳ StringSplitOptions.RemoveEmptyEntries);
189 var arrays = strings.Select(x => x.Select(y =>
    ↳ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
190 for (int i = 0; i < arrays.Length; i++)
191 {
192     unicodeSequences.Create(arrays[i].ShiftRight());
193 }
194
195 var linksCountAfterCreation = links.Count();
196
197 // get list of sequences links
198 // for each sequence link
199 //     create new sequence version
200 //     if new sequence is not the same as sequence link
201 //         delete sequence link
202 //         collect garbadge
203 unicodeSequences.CompactAll();
204
205 var linksCountAfterCompactification = links.Count();
206
207 Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
208 }
209 }
210 }
211 }

```

### 1.118 ./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4 using System.Linq;
5 using Xunit;
6 using Platform.Data.Sequences;
7 using Platform.Data.Doublets.Sequences.Converters;
8 using Platform.Data.Doublets.Sequences.Walkers;
9 using Platform.Data.Doublets.Sequences;
10
11 namespace Platform.Data.Doublets.Tests
12 {
13     public static class ReadSequenceTests
14     {
15         [Fact]
16         public static void ReadSequenceTest()
17         {
18             const long sequenceLength = 2000;
19
20             using (var scope = new TempLinksTestScope(useSequences: false))
21             {
22                 var links = scope.Links;
23                 var sequences = new Sequences.Sequences(links, new SequencesOptions

```

```

50         Assert.True(sequence.SequenceEqual(readSequence2));
51
52         // Assert.True(sw2.Elapsed < sw3.Elapsed);
53
54         Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
55         ↳ {sw2.Elapsed}");
56
57         for (var i = 0; i < sequenceLength; i++)
58         {
59             links.Delete(sequence[i]);
60         }
61     }
62 }
63 }

```

# 1.119 ./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs

```

1  using System.IO;
2  using Xunit;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using Platform.Data.Doublets.ResizableDirectMemory.Specific;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public static class ResizableDirectMemoryLinksTests
10     {
11         private static readonly LinksConstants<ulong> _constants =
12         ↳ Default<LinksConstants<ulong>>.Instance;
13
14         [Fact]
15         public static void BasicFileMappedMemoryTest()
16         {
17             var tempFilename = Path.GetTempFileName();
18             using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
19             {
20                 memoryAdapter.TestBasicMemoryOperations();
21             }
22             File.Delete(tempFilename);
23
24             [Fact]
25             public static void BasicHeapMemoryTest()
26             {
27                 using (var memory = new
28                 ↳ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
29                 using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
30                 ↳ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
31                 {
32                     memoryAdapter.TestBasicMemoryOperations();
33                 }
34
35                 private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
36                 {
37                     var link = memoryAdapter.Create();
38                     memoryAdapter.Delete(link);
39                 }
40
41                 [Fact]
42                 public static void NonexistentReferencesHeapMemoryTest()
43                 {
44                     using (var memory = new
45                     ↳ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
46                     using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
47                     ↳ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
48                     {
49                         memoryAdapter.TestNonexistentReferences();
50                     }
51
52                     private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
53                     {
54                         var link = memoryAdapter.Create();
55                         memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
56                         var resultLink = _constants.Null;
57                         memoryAdapter.Each(foundLink =>
58                         {
59                             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 }

```

### 1.120 ./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.ResizableDirectMemory.Generic;
7  using Platform.Data.Doublets.ResizableDirectMemory.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<UInt64ResizableDirectMemoryLinks>();
31                 var instance = scope.Use<ILinks<ulong>>();
32                 Assert.IsType<UInt64ResizableDirectMemoryLinks>(instance);
33             }
34         }
35
36         [Fact]
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,
50 ↵ ResizableDirectMemoryLinks<ulong>>>())
51             {
52                 var links = scope.Use<ILinks<ulong>>();
53                 Assert.IsType<ResizableDirectMemoryLinks<ulong>>(links);
54             }
55         }
56     }
57 }

```

### 1.121 ./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 =
22             ↳ Default<LinksConstants<ulong>>.Instance;
23
24         static SequencesTests()
25         {
26             // Trigger static constructor to not mess with performance measurements
27             _ = BitString.GetBitMaskFromIndex(1);
28         }
29
30         [Fact]
31         public static void CreateAllVariantsTest()
32         {
33             const long sequenceLength = 8;
34
35             using (var scope = new TempLinksTestScope(useSequences: true))
36             {
37                 var links = scope.Links;
38                 var sequences = scope.Sequences;
39
40                 var sequence = new ulong[sequenceLength];
41                 for (var i = 0; i < sequenceLength; i++)
42                 {
43                     sequence[i] = links.Create();
44                 }
45
46                 var sw1 = Stopwatch.StartNew();
47                 var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
48
49                 var sw2 = Stopwatch.StartNew();
50                 var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
51
52                 Assert.True(results1.Count > results2.Length);
53                 Assert.True(sw1.Elapsed > sw2.Elapsed);
54
55                 for (var i = 0; i < sequenceLength; i++)
56                 {
57                     links.Delete(sequence[i]);
58                 }
59
60                 Assert.True(links.Count() == 0);
61             }
62         }
63
64         //[Fact]
65         //public void CUDTest()
66         //{
67             // var tempFilename = Path.GetTempFileName();
68
69             // const long sequenceLength = 8;
70
71             // const ulong itself = LinksConstants.Itself;
72
73             // using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
74             // ↳ DefaultLinksSizeStep))
75             // using (var links = new Links(memoryAdapter))
76             // {
77                 // var sequence = new ulong[sequenceLength];
78                 // for (var i = 0; i < sequenceLength; i++)
79                 //     sequence[i] = links.Create(itself, itself);
80
81                 // SequencesOptions o = new SequencesOptions();
82
83                 // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
84                 // o.
85
86                 // var sequences = new Sequences(links);
87
88                 // var sw1 = Stopwatch.StartNew();
89                 // var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
90             }
91         }
92     }
93 }

```

```

89 //         var sw2 = Stopwatch.StartNew();
90 //         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
91
92 //         Assert.True(results1.Count > results2.Length);
93 //         Assert.True(sw1.Elapsed > sw2.Elapsed);
94
95 //         for (var i = 0; i < sequenceLength; i++)
96 //             links.Delete(sequence[i]);
97 //     }
98
99 //     File.Delete(tempFilename);
100 // }
101
102 [Fact]
103 public static void AllVariantsSearchTest()
104 {
105     const long sequenceLength = 8;
106
107     using (var scope = new TempLinksTestScope(useSequences: true))
108     {
109         var links = scope.Links;
110         var sequences = scope.Sequences;
111
112         var sequence = new ulong[sequenceLength];
113         for (var i = 0; i < sequenceLength; i++)
114         {
115             sequence[i] = links.Create();
116         }
117
118         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119
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 ==
192         ↪ searchResults3.First());
193
194     //Assert.True(sw1.Elapsed < sw2.Elapsed);
195
196     for (var i = 0; i < sequenceLength; i++)
197     {
198         links.Delete(sequence[i]);
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 + ": " +
221             ↪ sequences.FormatSequence(x)).ToList();
222         //Global.Trash = createResultsStrings;
223
224         var partialSequence = new ulong[sequenceLength - 2];
225
226         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
227
228         var sw1 = Stopwatch.StartNew();
229         var searchResults1 =
230             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
231
232         var sw2 = Stopwatch.StartNew();
233         var searchResults2 =
234             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
235
236         //var sw3 = Stopwatch.StartNew();
237         //var searchResults3 =
238             ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
239
240         var sw4 = Stopwatch.StartNew();
241         var searchResults4 =
242             ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
243
244         //Global.Trash = searchResults3;
245
246         //var searchResults1Strings = searchResults1.Select(x => x + ": " +
247             ↪ 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 =
286             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
287
288         var sw2 = Stopwatch.StartNew();
289         var searchResults2 =
290             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
291
292         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
293
294         Assert.True(searchResults2.Count == 1 && balancedVariant ==
295             ↪ searchResults2.First());
296
297         for (var i = 0; i < sequenceLength; i++)
298         {
299             links.Delete(sequence[i]);
300         }
301     }
302 }
303
304 [Fact(Skip = "Correct implementation is pending")]
305 public static void PatternMatchTest()
306 {
307     var zeroOrMany = Sequences.Sequences.ZeroOrMany;
308
309     using (var scope = new TempLinksTestScope(useSequences: true))
310     {
311         var links = scope.Links;
312         var sequences = scope.Sequences;
313
314         var e1 = links.Create();
315         var e2 = links.Create();
316
317         var sequence = new[]
318         {
319             e1, e2, e1, e2 // mama / papa
320         };
321     }
322 }

```

```

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 =
358         ↪ true }, useSequences: true))
359     {
360         var links = scope.Links;
361         var sequences = scope.Sequences;
362         var index = sequences.Options.Index;
363
364         var e1 = links.Create();
365         var e2 = links.Create();
366
367         var sequence = new[]
368         {
369             e1, e2, e1, e2 // mama / papa
370         };
371
372         Assert.False(index.MightContain(sequence));
373
374         index.Add(sequence);
375
376         Assert.True(index.MightContain(sequence));
377     }
378 }
379
380 /// <summary>Imported from https://raw.githubusercontent.com/Konard/LinksPlatform/%
381 ↪ 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
382 ↪ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
383 private static readonly string _exampleText =
384     @"([english
385     ↪ version] (https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

```

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?

[![чёрное пространство, белое  
 ↪ пространство] (<https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png>  
 ↪ "чёрное пространство, белое пространство")] (<https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png>)

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  
401 [![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)

402  
403 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?

404  
405 [![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)

406  
407 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?

408  
409 [![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)

410  
411 Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она  
→ может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть  
→ граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два  
→ объекта, как бы это выглядело?

412  
413 [![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)

414  
415 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?

416  
417 [![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)

418  
419 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?

420

```

421  [![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)
422
423  На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
      ↳ рекурсии или фрактала?
424
425  [![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)
426
427  Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
      ↳ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
428
429  [![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)
430
431  ...
432
433  [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anim
      ↳ ation-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)
482     ↪ == sequence[0]);
483 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
484     ↪ == sequence[1]);
485 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
486     ↪ == sequence[2]);
487 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
488     ↪ == sequence[3]);
489
490 }
491
492 [Fact]
493 public static void CompressionEfficiencyTest()
494 {
495     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
496     ↪ StringSplitOptions.RemoveEmptyEntries);
497     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
498     var totalCharacters = arrays.Select(x => x.Length).Sum();
499
500     using (var scope1 = new TempLinksTestScope(useSequences: true))
501     using (var scope2 = new TempLinksTestScope(useSequences: true))
502     using (var scope3 = new TempLinksTestScope(useSequences: true))
503     {
504         scope1.Links.Unsync.UseUnicode();
505         scope2.Links.Unsync.UseUnicode();
506         scope3.Links.Unsync.UseUnicode();
507
508         var balancedVariantConverter1 = new
509         ↪ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
510         var totalSequenceSymbolFrequencyCounter = new
511         ↪ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
512         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
513         ↪ totalSequenceSymbolFrequencyCounter);
514         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
515         ↪ balancedVariantConverter1, linkFrequenciesCache1,
516         ↪ doInitialFrequenciesIncrement: false);
517
518         //var compressor2 = scope2.Sequences;
519         var compressor3 = scope3.Sequences;
520
521         var constants = Default<LinksConstants<ulong>>.Instance;
522
523         var sequences = compressor3;
524         //var meaningRoot = links.CreatePoint();
525         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
526         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
527         //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
528         ↪ constants.Itself);
529
530         //var unaryNumberToAddressConverter = new
531         ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
532         //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
533         ↪ unaryOne);
534         //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
535         ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
536         //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
537         ↪ frequencyPropertyMarker, frequencyMarker);
538         //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
539         ↪ frequencyPropertyOperator, frequencyIncrementer);
540         //var linkToItsFrequencyNumberConverter = new
541         ↪ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
542         ↪ unaryNumberToAddressConverter);
543
544         var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
545         ↪ totalSequenceSymbolFrequencyCounter);
546
547         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque_
548         ↪ ncyNumberConverter<ulong>(linkFrequenciesCache3);

```

```

530     var sequenceToItsLocalElementLevelsConverter = new
531         ↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
532         ↳ linkToItsFrequencyNumberConverter);
533     var optimalVariantConverter = new
534         ↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
535         ↳ sequenceToItsLocalElementLevelsConverter);
536
537     var compressed1 = new ulong[arrays.Length];
538     var compressed2 = new ulong[arrays.Length];
539     var compressed3 = new ulong[arrays.Length];
540
541     var START = 0;
542     var END = arrays.Length;
543
544     //for (int i = START; i < END; i++)
545     //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
546
547     var initialCount1 = scope2.Links.Unsync.Count();
548
549     var sw1 = Stopwatch.StartNew();
550
551     for (int i = START; i < END; i++)
552     {
553         linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
554         compressed1[i] = compressor1.Convert(arrays[i]);
555     }
556
557     var elapsed1 = sw1.Elapsed;
558
559     var balancedVariantConverter2 = new
560         ↳ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
561
562     var initialCount2 = scope2.Links.Unsync.Count();
563
564     var sw2 = Stopwatch.StartNew();
565
566     for (int i = START; i < END; i++)
567     {
568         compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
569     }
570
571     var elapsed2 = sw2.Elapsed;
572
573     for (int i = START; i < END; i++)
574     {
575         linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
576     }
577
578     var initialCount3 = scope3.Links.Unsync.Count();
579
580     var sw3 = Stopwatch.StartNew();
581
582     for (int i = START; i < END; i++)
583     {
584         //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
585         compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
586     }
587
588     var elapsed3 = sw3.Elapsed;
589
590     Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
591         ↳ Optimal variant: {elapsed3}");
592
593     // Assert.True(elapsed1 > elapsed2);
594
595     // Checks
596     for (int i = START; i < END; i++)
597     {
598         var sequence1 = compressed1[i];
599         var sequence2 = compressed2[i];
600         var sequence3 = compressed3[i];
601
602         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
603             ↳ scope1.Links.Unsync);
604
605         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
606             ↳ scope2.Links.Unsync);
607
608         var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
609             ↳ scope3.Links.Unsync);

```

```

601
602     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
        ↳ link.IsPartialPoint());
603     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
        ↳ link.IsPartialPoint());
604     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
        ↳ link.IsPartialPoint());
605
606     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
        ↳ arrays[i].Length > 3)
607     //    Assert.False(structure1 == structure2);
608     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
        ↳ arrays[i].Length > 3)
609     //    Assert.False(structure3 == structure2);
610
611     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
612     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
613 }
614
615 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
        ↳ totalCharacters);
616 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
        ↳ totalCharacters);
617 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
        ↳ totalCharacters);
618
619 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
        ↳ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
        ↳ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
        ↳ totalCharacters}");
620
621 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
        ↳ scope2.Links.Unsync.Count() - initialCount2);
622 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
        ↳ scope2.Links.Unsync.Count() - initialCount2);
623
624 var duplicateProvider1 = new
        ↳ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
625 var duplicateProvider2 = new
        ↳ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
626 var duplicateProvider3 = new
        ↳ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
627
628 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
629 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
630 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
631
632 var duplicates1 = duplicateCounter1.Count();
633
634 ConsoleHelpers.Debug("-----");
635
636 var duplicates2 = duplicateCounter2.Count();
637
638 ConsoleHelpers.Debug("-----");
639
640 var duplicates3 = duplicateCounter3.Count();
641
642 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
643
644 linkFrequenciesCache1.ValidateFrequencies();
645 linkFrequenciesCache3.ValidateFrequencies();
646 }
647 }
648
649 [Fact]
650 public static void CompressionStabilityTest()
651 {
652     // TODO: Fix bug (do a separate test)
653     //const ulong minNumbers = 0;
654     //const ulong maxNumbers = 1000;
655
656     const ulong minNumbers = 10000;
657     const ulong maxNumbers = 12500;
658
659     var strings = new List<string>();
660
661     for (ulong i = minNumbers; i < maxNumbers; i++)
662     {

```

```

663         strings.Add(i.ToString());
664     }
665
666     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
667     var totalCharacters = arrays.Select(x => x.Length).Sum();
668
669     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
        ↳ SequencesOptions<ulong> { UseCompression = true,
        ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
        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)
            // 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);

```



```

740 // Checks
741 for (int i = START; i < END; i++)
742 {
743     var sequence1 = compressed1[i];
744     var sequence2 = compressed2[i];
745
746     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
747     {
748         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
749             ↪ scope1.Links);
750
751         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
752             ↪ scope2.Links);
753
754         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
755             ↪ link.IsPartialPoint());
756         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
757             ↪ link.IsPartialPoint());
758
759         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
760             ↪ arrays[i].Length > 3)
761         //    Assert.False(structure1 == structure2);
762
763         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
764     }
765 }
766
767 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
768 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
769
770 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
771     ↪ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
772     ↪ totalCharacters}");
773
774 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
775
776 //compressor1.ValidateFrequencies();
777 }
778 }
779
780 [Fact]
781 public static void RandomNumbersCompressionQualityTest()
782 {
783     const ulong N = 500;
784
785     //const ulong minNumbers = 10000;
786     //const ulong maxNumbers = 20000;
787
788     //var strings = new List<string>();
789
790     //for (ulong i = 0; i < N; i++)
791     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
792         ↪ maxNumbers).ToString());
793
794     var strings = new List<string>();
795
796     for (ulong i = 0; i < N; i++)
797     {
798         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
799     }
800
801     strings = strings.Distinct().ToList();
802
803     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
804     var totalCharacters = arrays.Select(x => x.Length).Sum();
805
806     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
807         ↪ SequencesOptions<ulong> { UseCompression = true,
808         ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
809     using (var scope2 = new TempLinksTestScope(useSequences: true))
810     {
811         scope1.Links.UseUnicode();
812         scope2.Links.UseUnicode();
813
814         var compressor1 = scope1.Sequences;
815         var compressor2 = scope2.Sequences;
816
817         var compressed1 = new ulong[arrays.Length];
818         var compressed2 = new ulong[arrays.Length];

```

```

809
810     var sw1 = Stopwatch.StartNew();
811
812     var START = 0;
813     var END = arrays.Length;
814
815     for (int i = START; i < END; i++)
816     {
817         compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
818     }
819
820     var elapsed1 = sw1.Elapsed;
821
822     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
823
824     var sw2 = Stopwatch.StartNew();
825
826     for (int i = START; i < END; i++)
827     {
828         compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
829     }
830
831     var elapsed2 = sw2.Elapsed;
832
833     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
834         ↪ {elapsed2}");
835
836     Assert.True(elapsed1 > elapsed2);
837
838     // Checks
839     for (int i = START; i < END; i++)
840     {
841         var sequence1 = compressed1[i];
842         var sequence2 = compressed2[i];
843
844         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
845         {
846             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
847                 ↪ scope1.Links);
848
849             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
850                 ↪ scope2.Links);
851
852             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
853         }
854     }
855
856     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
857     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
858
859     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
860         ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
861         ↪ totalCharacters}}");
862
863     // Can be worse than balanced variant
864     //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
865
866     //compressor1.ValidateFrequencies();
867 }
868
869 [Fact]
870 public static void AllTreeBreakDownAtSequencesCreationBugTest()
871 {
872     // Made out of AllPossibleConnectionsTest test.
873
874     //const long sequenceLength = 5; //100% bug
875     const long sequenceLength = 4; //100% bug
876     //const long sequenceLength = 3; //100% _no_bug (ok)
877
878     using (var scope = new TempLinksTestScope(useSequences: true))
879     {
880         var links = scope.Links;
881         var sequences = scope.Sequences;
882
883         var sequence = new ulong[sequenceLength];
884         for (var i = 0; i < sequenceLength; i++)
885         {
886             sequence[i] = links.Create();
887         }
888     }
889 }

```

```

883     }
884
885     var createResults = sequences.CreateAllVariants2(sequence);
886
887     Global.Trash = createResults;
888
889     for (var i = 0; i < sequenceLength; i++)
890     {
891         links.Delete(sequence[i]);
892     }
893 }
894
895 [Fact]
896 public static void AllPossibleConnectionsTest()
897 {
898     const long sequenceLength = 5;
899
900     using (var scope = new TempLinksTestScope(useSequences: true))
901     {
902         var links = scope.Links;
903         var sequences = scope.Sequences;
904
905         var sequence = new ulong[sequenceLength];
906         for (var i = 0; i < sequenceLength; i++)
907         {
908             sequence[i] = links.Create();
909         }
910
911         var createResults = sequences.CreateAllVariants2(sequence);
912         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
913
914         for (var i = 0; i < 1; i++)
915         {
916             var sw1 = Stopwatch.StartNew();
917             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
918
919             var sw2 = Stopwatch.StartNew();
920             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
921
922             var sw3 = Stopwatch.StartNew();
923             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
924
925             var sw4 = Stopwatch.StartNew();
926             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
927
928             Global.Trash = searchResults3;
929             Global.Trash = searchResults4; //-V3008
930
931             var intersection1 = createResults.Intersect(searchResults1).ToList();
932             Assert.True(intersection1.Count == createResults.Length);
933
934             var intersection2 = reverseResults.Intersect(searchResults1).ToList();
935             Assert.True(intersection2.Count == reverseResults.Length);
936
937             var intersection0 = searchResults1.Intersect(searchResults2).ToList();
938             Assert.True(intersection0.Count == searchResults2.Count);
939
940             var intersection3 = searchResults2.Intersect(searchResults3).ToList();
941             Assert.True(intersection3.Count == searchResults3.Count);
942
943             var intersection4 = searchResults3.Intersect(searchResults4).ToList();
944             Assert.True(intersection4.Count == searchResults4.Count);
945         }
946
947         for (var i = 0; i < sequenceLength; i++)
948         {
949             links.Delete(sequence[i]);
950         }
951     }
952 }
953
954 [Fact(Skip = "Correct implementation is pending")]
955 public static void CalculateAllUsagesTest()
956 {
957     const long sequenceLength = 3;
958
959     using (var scope = new TempLinksTestScope(useSequences: true))
960     {
961         var links = scope.Links;
962

```

```

963     var sequences = scope.Sequences;
964
965     var sequence = new ulong[sequenceLength];
966     for (var i = 0; i < sequenceLength; i++)
967     {
968         sequence[i] = links.Create();
969     }
970
971     var createResults = sequences.CreateAllVariants2(sequence);
972
973     //var reverseResults =
974     ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
975
976     for (var i = 0; i < 1; i++)
977     {
978         var linksTotalUsages1 = new ulong[links.Count() + 1];
979
980         sequences.CalculateAllUsages(linksTotalUsages1);
981
982         var linksTotalUsages2 = new ulong[links.Count() + 1];
983
984         sequences.CalculateAllUsages2(linksTotalUsages2);
985
986         var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
987         Assert.True(intersection1.Count == linksTotalUsages2.Length);
988     }
989
990     for (var i = 0; i < sequenceLength; i++)
991     {
992         links.Delete(sequence[i]);
993     }
994 }
995 }
996 }

```

### 1.122 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs

```

1  using System;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Memory;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.SplitMemory.Generic;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public unsafe static class SplitMemoryGenericLinksTests
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(Skip = "Common trees index is required for linking non-existent references")]
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
34             ↪ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
35             ↪ implementation of tree cuts out 5 bits from the address space.
36             Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
37             ↪ stMultipleRandomCreationsAndDeletions(100));
38             Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
39             ↪ MultipleRandomCreationsAndDeletions(100));
40             Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
41             ↪ tMultipleRandomCreationsAndDeletions(100));
42         }
43     }
44 }

```

```

38
39     private static void Using<TLink>(Action<ILinks<TLink>>> action)
40     {
41         //using (var scope = new Scope<Types<HeapResizableDirectMemory,
42         ↪ SplitMemoryLinks<TLink>>>())
43         using (var dataMemory = new HeapResizableDirectMemory())
44         using (var indexMemory = new HeapResizableDirectMemory())
45         using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory))
46         {
47             action(memory);
48         }
49     }
50 }

```

### 1.123 ./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.ResizableDirectMemory.Specific;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public class TempLinksTestScope : DisposableBase
10     {
11         public ILinks<ulong> MemoryAdapter { get; }
12         public SynchronizedLinks<ulong> Links { get; }
13         public Sequences.Sequences Sequences { get; }
14         public string TempFilename { get; }
15         public string TempTransactionLogFilename { get; }
16         private readonly bool _deleteFiles;
17
18         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
19         ↪ useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
20         ↪ useLog) { }
21
22         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
23         ↪ true, bool useSequences = false, bool useLog = false)
24         {
25             _deleteFiles = deleteFiles;
26             TempFilename = Path.GetTempFileName();
27             TempTransactionLogFilename = Path.GetTempFileName();
28             var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
29             MemoryAdapter = useLog ? (ILinks<ulong>)new
30             ↪ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
31             ↪ coreMemoryAdapter;
32             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
33             if (useSequences)
34             {
35                 Sequences = new Sequences.Sequences(Links, sequencesOptions);
36             }
37         }
38
39         protected override void Dispose(bool manual, bool wasDisposed)
40         {
41             if (!wasDisposed)
42             {
43                 Links.Unsync.DisposeIfPossible();
44                 if (_deleteFiles)
45                 {
46                     DeleteFiles();
47                 }
48             }
49         }
50
51         public void DeleteFiles()
52         {
53             File.Delete(TempFilename);
54             File.Delete(TempTransactionLogFilename);
55         }
56     }
57 }

```

### 1.124 ./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs

```

1  using System.Collections.Generic;
2  using Xunit;
3  using Platform.Ranges;

```

```

4  using Platform.Numbers;
5  using Platform.Random;
6  using Platform.Setters;
7  using Platform.Converters;
8
9  namespace Platform.Data.Doublets.Tests
10 {
11     public static class TestExtensions
12     {
13         public static void TestCRUDOperations<T>(this ILinks<T> links)
14         {
15             var constants = links.Constants;
16
17             var equalityComparer = EqualityComparer<T>.Default;
18
19             var zero = default(T);
20             var one = Arithmetic.Increment(zero);
21
22             // Create Link
23             Assert.True(equalityComparer.Equals(links.Count(), zero));
24
25             var setter = new Setter<T>(constants.Null);
26             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
27
28             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
29
30             var linkAddress = links.Create();
31
32             var link = new Link<T>(links.GetLink(linkAddress));
33
34             Assert.True(link.Count == 3);
35             Assert.True(equalityComparer.Equals(link.Index, linkAddress));
36             Assert.True(equalityComparer.Equals(link.Source, constants.Null));
37             Assert.True(equalityComparer.Equals(link.Target, constants.Null));
38
39             Assert.True(equalityComparer.Equals(links.Count(), one));
40
41             // Get first link
42             setter = new Setter<T>(constants.Null);
43             links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
44
45             Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
46
47             // Update link to reference itself
48             links.Update(linkAddress, linkAddress, linkAddress);
49
50             link = new Link<T>(links.GetLink(linkAddress));
51
52             Assert.True(equalityComparer.Equals(link.Source, linkAddress));
53             Assert.True(equalityComparer.Equals(link.Target, linkAddress));
54
55             // Update link to reference null (prepare for delete)
56             var updated = links.Update(linkAddress, constants.Null, constants.Null);
57
58             Assert.True(equalityComparer.Equals(updated, linkAddress));
59
60             link = new Link<T>(links.GetLink(linkAddress));
61
62             Assert.True(equalityComparer.Equals(link.Source, constants.Null));
63             Assert.True(equalityComparer.Equals(link.Target, constants.Null));
64
65             // Delete link
66             links.Delete(linkAddress);
67
68             Assert.True(equalityComparer.Equals(links.Count(), zero));
69
70             setter = new Setter<T>(constants.Null);
71             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
72
73             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
74         }
75
76         public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
77         {
78             // Constants
79             var constants = links.Constants;
80             var equalityComparer = EqualityComparer<T>.Default;
81
82             var zero = default(T);
83             var one = Arithmetic.Increment(zero);
84             var two = Arithmetic.Increment(one);

```

```

85
86     var h106E = new Hybrid<T>(106L, isExternal: true);
87     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
88     var h108E = new Hybrid<T>(-108L);
89
90     Assert.Equal(106L, h106E.AbsoluteValue);
91     Assert.Equal(107L, h107E.AbsoluteValue);
92     Assert.Equal(108L, h108E.AbsoluteValue);
93
94     // Create Link (External -> External)
95     var linkAddress1 = links.Create();
96
97     links.Update(linkAddress1, h106E, h108E);
98
99     var link1 = new Link<T>(links.GetLink(linkAddress1));
100
101     Assert.True(equalityComparer.Equals(link1.Source, h106E));
102     Assert.True(equalityComparer.Equals(link1.Target, h108E));
103
104     // Create Link (Internal -> External)
105     var linkAddress2 = links.Create();
106
107     links.Update(linkAddress2, linkAddress1, h108E);
108
109     var link2 = new Link<T>(links.GetLink(linkAddress2));
110
111     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
112     Assert.True(equalityComparer.Equals(link2.Target, h108E));
113
114     // Create Link (Internal -> Internal)
115     var linkAddress3 = links.Create();
116
117     links.Update(linkAddress3, linkAddress1, linkAddress2);
118
119     var link3 = new Link<T>(links.GetLink(linkAddress3));
120
121     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
122     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
123
124     // Search for created link
125     var setter1 = new Setter<T>(constants.Null);
126     links.Each(h106E, h108E, setter1.SetAndReturnFalse);
127
128     Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
129
130     // Search for nonexistent link
131     var setter2 = new Setter<T>(constants.Null);
132     links.Each(h106E, h107E, setter2.SetAndReturnFalse);
133
134     Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
135
136     // Update link to reference null (prepare for delete)
137     var updated = links.Update(linkAddress3, constants.Null, constants.Null);
138
139     Assert.True(equalityComparer.Equals(updated, linkAddress3));
140
141     link3 = new Link<T>(links.GetLink(linkAddress3));
142
143     Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
144     Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
145
146     // Delete link
147     links.Delete(linkAddress3);
148
149     Assert.True(equalityComparer.Equals(links.Count(), two));
150
151     var setter3 = new Setter<T>(constants.Null);
152     links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
153
154     Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
155 }
156
157 public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
→ links, int maximumOperationsPerCycle)
158 {
159     var comparer = Comparer<TLink>.Default;
160     var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
161     var uint64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
162     for (var N = 1; N < maximumOperationsPerCycle; N++)
163     {

```

```

164     var random = new System.Random(N);
165     var created = OUL;
166     var deleted = OUL;
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 }

```

### 1.125 ./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;
19 using Platform.Memory;
20 using Platform.Data.Doublets.Decorators;
21 using Platform.Data.Doublets.ResizableDirectMemory.Specific;
22
23 namespace Platform.Data.Doublets.Tests
24 {
25     public static class UInt64LinksTests
26     {
27         private static readonly LinksConstants<ulong> _constants =
            ↪ Default<LinksConstants<ulong>>.Instance;
28
29         private const long Iterations = 10 * 1024;
30
31         #region Concept
32
33         [Fact]
34         public static void MultipleCreateAndDeleteTest()

```



```

35 {
36     using (var scope = new Scope<Types<HeapResizableDirectMemory,
    ↪ UInt64ResizableDirectMemoryLinks>>())
37     {
38         new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeleti
    ↪ ons(100);
39     }
40 }
41
42 [Fact]
43 public static void CascadeUpdateTest()
44 {
45     var itself = _constants.Itself;
46     using (var scope = new TempLinksTestScope(useLog: true))
47     {
48         var links = scope.Links;
49
50         var l1 = links.Create();
51         var l2 = links.Create();
52
53         l2 = links.Update(l2, l2, l1, l2);
54
55         links.CreateAndUpdate(l2, itself);
56         links.CreateAndUpdate(l2, itself);
57
58         l2 = links.Update(l2, l1);
59
60         links.Delete(l2);
61
62         Global.Trash = links.Count();
63
64         links.Unsync.DisposeIfPossible(); // Close links to access log
65
66         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
    ↪ e.TempTransactionLogFilename);
67     }
68 }
69
70 [Fact]
71 public static void BasicTransactionLogTest()
72 {
73     using (var scope = new TempLinksTestScope(useLog: true))
74     {
75         var links = scope.Links;
76         var l1 = links.Create();
77         var l2 = links.Create();
78
79         Global.Trash = links.Update(l2, l2, l1, l2);
80
81         links.Delete(l1);
82
83         links.Unsync.DisposeIfPossible(); // Close links to access log
84
85         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
    ↪ e.TempTransactionLogFilename);
86     }
87 }
88
89 [Fact]
90 public static void TransactionAutoRevertedTest()
91 {
92     // Auto Reverted (Because no commit at transaction)
93     using (var scope = new TempLinksTestScope(useLog: true))
94     {
95         var links = scope.Links;
96         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
97         using (var transaction = transactionsLayer.BeginTransaction())
98         {
99             var l1 = links.Create();
100             var l2 = links.Create();
101
102             links.Update(l2, l2, l1, l2);
103         }
104
105         Assert.Equal(0UL, links.Count());
106
107         links.Unsync.DisposeIfPossible();
108

```

```

109         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
110             ↪ cope.TempTransactionLogFilename);
111         Assert.Single(transitions);
112     }
113 }
114 [Fact]
115 public static void TransactionUserCodeErrorNoDataSavedTest()
116 {
117     // User Code Error (Autoreverted), no data saved
118     var itself = _constants.Itself;
119
120     TempLinksTestScope lastScope = null;
121     try
122     {
123         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
124             ↪ useLog: true))
125         {
126             var links = scope.Links;
127             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
128                 ↪ atorBase<ulong>)links.Unsync).Links;
129             using (var transaction = transactionsLayer.BeginTransaction())
130             {
131                 var l1 = links.CreateAndUpdate(itself, itself);
132                 var l2 = links.CreateAndUpdate(itself, itself);
133
134                 l2 = links.Update(l2, l2, l1, l2);
135
136                 links.CreateAndUpdate(l2, itself);
137                 links.CreateAndUpdate(l2, itself);
138
139                 //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
140                     ↪ tion>(scope.TempTransactionLogFilename);
141
142                 l2 = links.Update(l2, l1);
143
144                 links.Delete(l2);
145
146                 ExceptionThrower();
147
148                 transaction.Commit();
149             }
150             Global.Trash = links.Count();
151         }
152     }
153     catch
154     {
155         Assert.False(lastScope == null);
156
157         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
158             ↪ astScope.TempTransactionLogFilename);
159
160         Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
161             ↪ transitions[0].After.IsNull());
162
163         lastScope.DeleteFiles();
164     }
165 }
166 [Fact]
167 public static void TransactionUserCodeErrorSomeDataSavedTest()
168 {
169     // User Code Error (Autoreverted), some data saved
170     var itself = _constants.Itself;
171
172     TempLinksTestScope lastScope = null;
173     try
174     {
175         using (var scope = new TempLinksTestScope(useLog: true))
176         {
177             var links = scope.Links;
178             l1 = links.CreateAndUpdate(itself, itself);
179             l2 = links.CreateAndUpdate(itself, itself);
180
181             l2 = links.Update(l2, l2, l1, l2);
182

```

```

183         links.CreateAndUpdate(l2, itself);
184         links.CreateAndUpdate(l2, itself);
185
186         links.Unsync.DisposeIfPossible();
187
188         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
189             ↪ scope.TempTransactionLogFilename);
190     }
191
192     using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
193         ↪ useLog: true))
194     {
195         var links = scope.Links;
196         var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
197         using (var transaction = transactionsLayer.BeginTransaction())
198         {
199             l2 = links.Update(l2, l1);
200
201             links.Delete(l2);
202
203             ExceptionThrower();
204
205             transaction.Commit();
206         }
207
208         Global.Trash = links.Count();
209     }
210 }
211 catch
212 {
213     Assert.False(lastScope == null);
214
215     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
216         ↪ Scope.TempTransactionLogFilename);
217
218     lastScope.DeleteFiles();
219 }
220
221 [Fact]
222 public static void TransactionCommit()
223 {
224     var itself = _constants.Itself;
225
226     var tempDatabaseFilename = Path.GetTempFileName();
227     var tempTransactionLogFilename = Path.GetTempFileName();
228
229     // Commit
230     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
231         ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
232         ↪ tempTransactionLogFilename))
233     using (var links = new UInt64Links(memoryAdapter))
234     {
235         using (var transaction = memoryAdapter.BeginTransaction())
236         {
237             var l1 = links.CreateAndUpdate(itself, itself);
238             var l2 = links.CreateAndUpdate(itself, itself);
239
240             Global.Trash = links.Update(l2, l2, l1, l2);
241
242             links.Delete(l1);
243
244             transaction.Commit();
245         }
246
247         Global.Trash = links.Count();
248     }
249
250     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
251         ↪ sactionLogFilename);
252 }
253
254 [Fact]
255 public static void TransactionDamage()
256 {
257     var itself = _constants.Itself;
258
259     var tempDatabaseFilename = Path.GetTempFileName();
260     var tempTransactionLogFilename = Path.GetTempFileName();

```

```

256 // Commit
257 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
258     ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
259     ↳ tempTransactionLogFilename))
260 using (var links = new UInt64Links(memoryAdapter))
261 {
262     using (var transaction = memoryAdapter.BeginTransaction())
263     {
264         var l1 = links.CreateAndUpdate(itself, itself);
265         var l2 = links.CreateAndUpdate(itself, itself);
266
267         Global.Trash = links.Update(l2, l2, l1, l2);
268
269         links.Delete(l1);
270
271         transaction.Commit();
272     }
273     Global.Trash = links.Count();
274 }
275
276 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
277     ↳ sactionLogFilename);
278
279 // Damage database
280 FileHelpers.WriteFirst(tempTransactionLogFilename, new
281     ↳ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
282
283 // Try load damaged database
284 try
285 {
286     // TODO: Fix
287     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
288         ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
289         ↳ tempTransactionLogFilename))
290     using (var links = new UInt64Links(memoryAdapter))
291     {
292         Global.Trash = links.Count();
293     }
294 }
295 catch (NotSupportedException ex)
296 {
297     Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
298         ↳ yet.");
299 }
300
301 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
302     ↳ sactionLogFilename);
303
304 File.Delete(tempDatabaseFilename);
305 File.Delete(tempTransactionLogFilename);
306 }
307
308 [Fact]
309 public static void Bug1Test()
310 {
311     var tempDatabaseFilename = Path.GetTempFileName();
312     var tempTransactionLogFilename = Path.GetTempFileName();
313
314     var itself = _constants.Itself;
315
316     // User Code Error (Autoreverted), some data saved
317     try
318     {
319         ulong l1;
320         ulong l2;
321
322         using (var memory = new UInt64ResizableDirectMemoryLinks(tempDatabaseFilename))
323         using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
324             ↳ tempTransactionLogFilename))
325         using (var links = new UInt64Links(memoryAdapter))
326         {
327             l1 = links.CreateAndUpdate(itself, itself);
328             l2 = links.CreateAndUpdate(itself, itself);
329
330             l2 = links.Update(l2, l2, l1, l2);
331         }
332     }
333 }

```

```

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 UInt64ResizableDirectMemoryLinks(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.CheckPathExistence(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) ==
403         ↳ "(5:(4:5 (6:5 4)) 6)");
404     Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
405         ↳ "(6:(5:(4:5 6) 6) 4)");
406     Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
407         ↳ "(4:(5:4 (6:5 4)) 6)");
408
409     // TODO: Think how to build balanced syntax tree while formatting structure (eg.
410     ↳ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
411
412     Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
413         ↳ "{{5}{5}{4}{6}}");
414     Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
415         ↳ "{{5}{6}{6}{4}}");
416     Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
417         ↳ "{{4}{5}{4}{6}}");
418 }
419
420 private static void DefaultFormatter(StringBuilder sb, ulong link)
421 {
422     sb.Append(link.ToString());
423 }
424
425 #endregion
426
427 #region Performance
428
429 /*
430 public static void RunAllPerformanceTests()
431 {
432     try
433     {
434         links.TestLinksInSteps();
435     }
436     catch (Exception ex)
437     {
438         ex.WriteToConsole();
439     }
440
441     return;
442
443     try
444     {
445         //ThreadPool.SetMaxThreads(2, 2);
446
447         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
448         ↳ результат
449         // Также это дополнительно помогает в отладке
450         // Увеличивает вероятность попадания информации в кэши
451         for (var i = 0; i < 10; i++)
452         {
453             //0 - 10 ГБ
454             //Каждые 100 МБ срез цифр
455
456             //links.TestGetSourceFunction();
457             //links.TestGetSourceFunctionInParallel();
458             //links.TestGetTargetFunction();
459             //links.TestGetTargetFunctionInParallel();
460             links.Create64BillionLinks();
461
462             links.TestRandomSearchFixed();
463             //links.Create64BillionLinksInParallel();
464             links.TestEachFunction();
465             //links.TestForeach();
466             //links.TestParallelForeach();
467         }
468
469         links.TestDeletionOfAllLinks();
470
471     }
472     catch (Exception ex)
473     {
474         ex.WriteToConsole();
475     }
476 }*/
477
478 /*
479 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 searchMeasurements = 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             searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
497
498             Console.WriteLine("\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.WriteLine("\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],
↪ searchMeasurements[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,
↪ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
524         }
525
526         ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↪ links.Total);
527     }
528
529     private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
↪ amountToCreate)
530     {
531         for (long i = 0; i < amountToCreate; i++)
532             links.Create(0, 0);
533     }
534
535     private static TimeSpan GetBaseRandomLoopOverhead(long loops)
536     {
537         return Measure(() =>
538         {
539             ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
540             ulong result = 0;
541             for (long i = 0; i < loops; i++)
542             {
543                 var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544                 var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
545

```

```

546         result += maxValue + source + target;
547     }
548     Global.Trash = result;
549 });
550 }
551 */
552
553 [Fact(Skip = "performance test")]
554 public static void GetSourceTest()
555 {
556     using (var scope = new TempLinksTestScope())
557     {
558         var links = scope.Links;
559         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
560                               ↪ Iterations);
561
562         ulong counter = 0;
563
564         //var firstLink = links.First();
565         // Создаём одну связь, из которой будет производить считывание
566         var firstLink = links.Create();
567
568         var sw = Stopwatch.StartNew();
569
570         // Тестируем саму функцию
571         for (ulong i = 0; i < Iterations; i++)
572         {
573             counter += links.GetSource(firstLink);
574         }
575
576         var elapsedTime = sw.Elapsed;
577
578         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
579
580         // Удаляем связь, из которой производилось считывание
581         links.Delete(firstLink);
582
583         ConsoleHelpers.Debug(
584             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
585             ↪ second), counter result: {3}",
586             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
587     }
588 }
589
590 [Fact(Skip = "performance test")]
591 public static void GetSourceInParallel()
592 {
593     using (var scope = new TempLinksTestScope())
594     {
595         var links = scope.Links;
596         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
597                               ↪ parallel.", Iterations);
598
599         long counter = 0;
600
601         //var firstLink = links.First();
602         var firstLink = links.Create();
603
604         var sw = Stopwatch.StartNew();
605
606         // Тестируем саму функцию
607         Parallel.For(0, Iterations, x =>
608         {
609             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
610             //Interlocked.Increment(ref counter);
611         });
612
613         var elapsedTime = sw.Elapsed;
614
615         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
616
617         links.Delete(firstLink);
618
619         ConsoleHelpers.Debug(
620             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
621             ↪ second), counter result: {3}",
622             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
623     }
624 }

```



```

621 [Fact(Skip = "performance test")]
622 public static void TestGetTarget()
623 {
624     using (var scope = new TempLinksTestScope())
625     {
626         var links = scope.Links;
627         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
628             ↪ Iterations);
629
630         ulong counter = 0;
631
632         //var firstLink = links.First();
633         var firstLink = links.Create();
634
635         var sw = Stopwatch.StartNew();
636
637         for (ulong i = 0; i < Iterations; i++)
638         {
639             counter += links.GetTarget(firstLink);
640         }
641
642         var elapsedTime = sw.Elapsed;
643
644         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
645
646         links.Delete(firstLink);
647
648         ConsoleHelpers.Debug(
649             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
650             ↪ second), counter result: {3}",
651             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
652     }
653 }
654 [Fact(Skip = "performance test")]
655 public static void TestGetTargetInParallel()
656 {
657     using (var scope = new TempLinksTestScope())
658     {
659         var links = scope.Links;
660         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
661             ↪ parallel.", Iterations);
662
663         long counter = 0;
664
665         //var firstLink = links.First();
666         var firstLink = links.Create();
667
668         var sw = Stopwatch.StartNew();
669
670         Parallel.For(0, Iterations, x =>
671         {
672             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
673             //Interlocked.Increment(ref counter);
674         });
675
676         var elapsedTime = sw.Elapsed;
677
678         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
679
680         links.Delete(firstLink);
681
682         ConsoleHelpers.Debug(
683             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
684             ↪ second), counter result: {3}",
685             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
686     }
687 }
688 // TODO: Заполнить базу данных перед тестом
689 /*
690 [Fact]
691 public void TestRandomSearchFixed()
692 {
693     var tempFilename = Path.GetTempFileName();
694
695     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
696         ↪ 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: 0(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}",
↪ iterations, elapsedTime, (long)iterationsPerSecond, counter);
754         }
755     }
756
757     [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
758     public static void TestEach()
759     {
760         using (var scope = new TempLinksTestScope())
761         {
762             var links = scope.Links;
763
764             var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
765
766

```

```

767         ConsoleHelpers.Debug("Testing Each function.");
768
769         var sw = Stopwatch.StartNew();
770
771         links.Each(counter.IncrementAndReturnTrue);
772
773         var elapsedTime = sw.Elapsed;
774
775         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
776
777         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
↪         links per second)",
            counter, elapsedTime, (long)linksPerSecond);
778     }
779 }
780
781 /*
782 [Fact]
783 public static void TestForeach()
784 {
785     var tempFilename = Path.GetTempFileName();
786
787     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     DefaultLinksSizeStep))
788     {
789         ulong counter = 0;
790
791         ConsoleHelpers.Debug("Testing foreach through links.");
792
793         var sw = Stopwatch.StartNew();
794
795         //foreach (var link in links)
796         //{
797             //    counter++;
798         //}
799
800         var elapsedTime = sw.Elapsed;
801
802         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
803
804         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪     links per second)", counter, elapsedTime, (long)linksPerSecond);
805     }
806
807     File.Delete(tempFilename);
808 }
809 */
810
811 /*
812 [Fact]
813 public static void TestParallelForeach()
814 {
815     var tempFilename = Path.GetTempFileName();
816
817     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     DefaultLinksSizeStep))
818     {
819
820         long counter = 0;
821
822         ConsoleHelpers.Debug("Testing parallel foreach through links.");
823
824         var sw = Stopwatch.StartNew();
825
826         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
827         //{
828             //    Interlocked.Increment(ref counter);
829         //});
830
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
↪     {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
836     }
837
838     File.Delete(tempFilename);
839 }
840 */
841

```

```

842 [Fact(Skip = "performance test")]
843 public static void Create64BillionLinks()
844 {
845     using (var scope = new TempLinksTestScope())
846     {
847         var links = scope.Links;
848         var linksBeforeTest = links.Count();
849
850         long linksToCreate = 64 * 1024 * 1024 /
851             ↳ UInt64ResizableDirectMemoryLinks.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)",
869             ↳ linksCreated, elapsedTime,
870             (long)linksPerSecond);
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 /
884             ↳ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
885
886         ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
887
888         Parallel.For(0, linksToCreate, x => links.Create());
889
890         var elapsedTime = sw.Elapsed;
891
892         var linksCreated = links.Count() - linksBeforeTest;
893         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
894
895         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
896             ↳ linksCreated, elapsedTime,
897             (long)linksPerSecond);
898     }
899 }
900 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
901 public static void TestDeletionOfAllLinks()
902 {
903     using (var scope = new TempLinksTestScope())
904     {
905         var links = scope.Links;
906         var linksBeforeTest = links.Count();
907
908         ConsoleHelpers.Debug("Deleting all links");
909
910         var elapsedTime = Performance.Measure(links.DeleteAll);
911
912         var linksDeleted = linksBeforeTest - links.Count();
913         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
914
915         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
916             ↳ linksDeleted, elapsedTime,
917             (long)linksPerSecond);
918     }
919 }

```

```

917     }
918
919     #endregion
920 }
921 }

```

#### 1.126 ./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
19                     ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
20                 var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
21                     ↪ powerOf2ToUnaryNumberConverter);
22                 var random = new System.Random(0);
23                 ulong[] numbers = new ulong[N];
24                 ulong[] unaryNumbers = new ulong[N];
25                 for (int i = 0; i < N; i++)
26                 {
27                     numbers[i] = random.NextUInt64();
28                     unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
29                 }
30                 var fromUnaryNumberConverterUsingOrOperation = new
31                     ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
32                     ↪ powerOf2ToUnaryNumberConverter);
33                 var fromUnaryNumberConverterUsingAddOperation = new
34                     ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
35                 for (int i = 0; i < N; i++)
36                 {
37                     Assert.Equal(numbers[i],
38                         ↪ fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
39                     Assert.Equal(numbers[i],
40                         ↪ fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
41                 }
42             }
43         }
44     }
45 }

```

#### 1.127 ./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.ResizableDirectMemory.Generic;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class UnicodeConvertersTests
19     {
20         [Fact]
21         public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
22         {
23             using (var scope = new TempLinksTestScope())
24             {
25                 var links = scope.Links;

```

```

26     var meaningRoot = links.CreatePoint();
27     var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
28     var powerOf2ToUnaryNumberConverter = new
    ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
29     var addressToUnaryNumberConverter = new
    ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
30     var unaryNumberToAddressConverter = new
    ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
    ↪ powerOf2ToUnaryNumberConverter);
31     TestCharAndUnicodeSymbolConverters(links, meaningRoot,
    ↪ addressToUnaryNumberConverter, unaryNumberToAddressConverter);
32 }
33 }
34
35 [Fact]
36 public static void CharAndRawNumberUnicodeSymbolConvertersTest()
37 {
38     using (var scope = new Scope<Types<HeapResizableDirectMemory,
    ↪ ResizableDirectMemoryLinks<ulong>>>())
39     {
40         var links = scope.Use<ILinks<ulong>>>();
41         var meaningRoot = links.CreatePoint();
42         var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
43         var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
44         TestCharAndUnicodeSymbolConverters(links, meaningRoot,
    ↪ addressToRawNumberConverter, rawNumberToAddressConverter);
45     }
46 }
47
48 private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
    ↪ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
    ↪ numberToAddressConverter)
49 {
50     var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
51     var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
    ↪ addressToNumberConverter, unicodeSymbolMarker);
52     var originalCharacter = 'H';
53     var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
54     var unicodeSymbolCriterionMatcher = new UnicodeSymbolCriterionMatcher<ulong>(links,
    ↪ unicodeSymbolMarker);
55     var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
    ↪ numberToAddressConverter, unicodeSymbolCriterionMatcher);
56     var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
57     Assert.Equal(originalCharacter, resultingCharacter);
58 }
59
60 [Fact]
61 public static void StringAndUnicodeSequenceConvertersTest()
62 {
63     using (var scope = new TempLinksTestScope())
64     {
65         var links = scope.Links;
66
67         var itself = links.Constants.Itself;
68
69         var meaningRoot = links.CreatePoint();
70         var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
71         var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
72         var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
73         var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
74         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
75
76         var powerOf2ToUnaryNumberConverter = new
    ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
77         var addressToUnaryNumberConverter = new
    ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
78         var charToUnicodeSymbolConverter = new
    ↪ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
    ↪ unicodeSymbolMarker);
79
80         var unaryNumberToAddressConverter = new
    ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
    ↪ powerOf2ToUnaryNumberConverter);
81         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
82         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
83         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
    ↪ frequencyPropertyMarker, frequencyMarker);

```

```

84     var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
85         ↪ frequencyPropertyOperator, frequencyIncrementer);
86     var linkToItsFrequencyNumberConverter = new
87         ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
88         ↪ unaryNumberToAddressConverter);
89     var sequenceToItsLocalElementLevelsConverter = new
90         ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
91         ↪ linkToItsFrequencyNumberConverter);
92     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
93         ↪ sequenceToItsLocalElementLevelsConverter);
94
95     var stringToUnicodeSequenceConverter = new
96         ↪ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
97         ↪ index, optimalVariantConverter, unicodeSequenceMarker);
98
99     var originalString = "Hello";
100
101     var unicodeSequenceLink =
102         ↪ stringToUnicodeSequenceConverter.Convert(originalString);
103
104     var unicodeSymbolCriterionMatcher = new
105         ↪ UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
106     var unicodeSymbolToCharConverter = new
107         ↪ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
108         ↪ unicodeSymbolCriterionMatcher);
109
110     var unicodeSequenceCriterionMatcher = new
111         ↪ UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
112
113     var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
114         ↪ unicodeSymbolCriterionMatcher.IsMatched);
115
116     var unicodeSequenceToStringConverter = new
117         ↪ UnicodeSequenceToStringConverter<ulong>(links,
118         ↪ unicodeSequenceCriterionMatcher, sequenceWalker,
119         ↪ unicodeSymbolToCharConverter);
120
121     var resultingString =
122         ↪ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
123
124     Assert.Equal(originalString, resultingString);
125 }
126 }
127 }
128 }
129 }
130 }

```

## Index

`./csharp/Platform.Data.Doublets.Tests/ComparisonTests.cs`, 164  
`./csharp/Platform.Data.Doublets.Tests/EqualityTests.cs`, 165  
`./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs`, 167  
`./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs`, 167  
`./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs`, 168  
`./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs`, 171  
`./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs`, 172  
`./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs`, 173  
`./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs`, 173  
`./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs`, 188  
`./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs`, 189  
`./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs`, 189  
`./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs`, 192  
`./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs`, 205  
`./csharp/Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs`, 205  
`./csharp/Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs`, 1  
`./csharp/Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs`, 1  
`./csharp/Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs`, 1  
`./csharp/Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs`, 2  
`./csharp/Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs`, 3  
`./csharp/Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs`, 3  
`./csharp/Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs`, 4  
`./csharp/Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs`, 4  
`./csharp/Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs`, 4  
`./csharp/Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs`, 5  
`./csharp/Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs`, 5  
`./csharp/Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs`, 6  
`./csharp/Platform.Data.Doublets/Decorators/UInt64Links.cs`, 6  
`./csharp/Platform.Data.Doublets/Decorators/UniLinks.cs`, 7  
`./csharp/Platform.Data.Doublets/Doublet.cs`, 12  
`./csharp/Platform.Data.Doublets/DoubletComparer.cs`, 12  
`./csharp/Platform.Data.Doublets/ILinks.cs`, 13  
`./csharp/Platform.Data.Doublets/ILinksExtensions.cs`, 13  
`./csharp/Platform.Data.Doublets/ISynchronizedLinks.cs`, 24  
`./csharp/Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs`, 25  
`./csharp/Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs`, 25  
`./csharp/Platform.Data.Doublets/Link.cs`, 26  
`./csharp/Platform.Data.Doublets/LinkExtensions.cs`, 29  
`./csharp/Platform.Data.Doublets/LinksOperatorBase.cs`, 29  
`./csharp/Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs`, 29  
`./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConverter.cs`, 30  
`./csharp/Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs`, 31  
`./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs`, 31  
`./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs`, 32  
`./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs`, 33  
`./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs`, 34  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksAvlBalancedTreeMethodsBase.cs`, 35  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSizeBalancedTreeMethodsBase.cs`, 39  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesAvlBalancedTreeMethods.cs`, 42  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesSizeBalancedTreeMethods.cs`, 43  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsAvlBalancedTreeMethods.cs`, 44  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsSizeBalancedTreeMethods.cs`, 46  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinks.cs`, 46  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinksBase.cs`, 48  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/UnusedLinksListMethods.cs`, 55  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/ILinksListMethods.cs`, 56  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.cs`, 56  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs`, 56  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/RawLink.cs`, 57  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs`, 58  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs`, 59  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs`, 60  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs`, 62  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs`, 63  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs`, 64  
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64ResizableDirectMemoryLinks.cs`, 65



./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64UnusedLinksListMethods.cs, 67  
./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 67  
./csharp/Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 68  
./csharp/Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 71  
./csharp/Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 71  
./csharp/Platform.Data.Doublets/Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 73  
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs, 73  
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs, 73  
./csharp/Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 74  
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 75  
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 75  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 78  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 79  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 80  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 80  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 80  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 81  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 82  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 82  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 82  
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 83  
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 84  
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 84  
./csharp/Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 84  
./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 85  
./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 86  
./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 86  
./csharp/Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 87  
./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 88  
./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 88  
./csharp/Platform.Data.Doublets/Sequences/Sequences.cs, 115  
./csharp/Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 126  
./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs, 126  
./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 129  
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 129  
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 130  
./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 132  
./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 132  
./csharp/Platform.Data.Doublets/SplitMemory/Generic/LinksSizeBalancedTreeMethodsBase.cs, 133  
./csharp/Platform.Data.Doublets/SplitMemory/Generic/LinksSourcesSizeBalancedTreeMethods.cs, 135  
./csharp/Platform.Data.Doublets/SplitMemory/Generic/LinksTargetsSizeBalancedTreeMethods.cs, 136  
./csharp/Platform.Data.Doublets/SplitMemory/Generic/SplitMemoryLinks.cs, 137  
./csharp/Platform.Data.Doublets/SplitMemory/Generic/SplitMemoryLinksBase.cs, 138  
./csharp/Platform.Data.Doublets/SplitMemory/Generic/UnusedLinksListMethods.cs, 146  
./csharp/Platform.Data.Doublets/SplitMemory/ILinksListMethods.cs, 147  
./csharp/Platform.Data.Doublets/SplitMemory/ILinksTreeMethods.cs, 147  
./csharp/Platform.Data.Doublets/SplitMemory/LinksHeaderIndexPart.cs, 147  
./csharp/Platform.Data.Doublets/SplitMemory/RawLinkDataPart.cs, 148  
./csharp/Platform.Data.Doublets/SplitMemory/RawLinkIndexPart.cs, 149  
./csharp/Platform.Data.Doublets/Stacks/Stack.cs, 149  
./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs, 150  
./csharp/Platform.Data.Doublets/SynchronizedLinks.cs, 150  
./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs, 151  
./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 153  
./csharp/Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 159  
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 159  
./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs, 160  
./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 163  
./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 163  
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 164  
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 164