

LinksPlatform's Platform.Data.Doublets Class Library

1.1 ./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 ./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 ./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        private ILinks<TLink> _facade;
12
13        public LinksConstants<TLink> Constants
14        {
15            [MethodImpl(MethodImplOptions.AggressiveInlining)]
16            get;
17        }
18
19        public ILinks<TLink> Facade
20        {
21            [MethodImpl(MethodImplOptions.AggressiveInlining)]
22            get => _facade;
23            [MethodImpl(MethodImplOptions.AggressiveInlining)]
24            set
25            {
26                _facade = value;
27            }
28        }
29    }
```

```

27         if (Links is LinksDecoratorBase<TLink> decorator)
28         {
29             decorator.Facade = value;
30         }
31         else if (Links is LinksDisposableDecoratorBase<TLink> disposableDecorator)
32         {
33             disposableDecorator.Facade = value;
34         }
35     }
36 }
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
40 {
41     Constants = links.Constants;
42     Facade = this;
43 }
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 public virtual TLink Count(IList<TLink> restrictions) => Links.Count(restrictions);
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
50     => Links.Each(handler, restrictions);
51
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 public virtual TLink Create(IList<TLink> restrictions) => Links.Create(restrictions);
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
57     Links.Update(restrictions, substitution);
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Disposables;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Decorators
9 {
10     public abstract class LinksDisposableDecoratorBase<TLink> : DisposableBase, ILinks<TLink>
11     {
12         private ILinks<TLink> _facade;
13
14         public LinksConstants<TLink> Constants
15         {
16             [MethodImpl(MethodImplOptions.AggressiveInlining)]
17             get;
18         }
19
20         public ILinks<TLink> Links
21         {
22             [MethodImpl(MethodImplOptions.AggressiveInlining)]
23             get;
24         }
25
26         public ILinks<TLink> Facade
27         {
28             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29             get => _facade;
30             [MethodImpl(MethodImplOptions.AggressiveInlining)]
31             set
32             {
33                 _facade = value;
34                 if (Links is LinksDecoratorBase<TLink> decorator)
35                 {
36                     decorator.Facade = value;
37                 }
38                 else if (Links is LinksDisposableDecoratorBase<TLink> disposableDecorator)
39                 {
40                     disposableDecorator.Facade = value;
41                 }
42             }
43         }
44     }
45 }

```

```

43     }
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected LinksDisposableDecoratorBase(ILinks<TLink> links)
47     {
48         Links = links;
49         Constants = links.Constants;
50         Facade = this;
51     }
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     public virtual TLink Count(IList<TLink> restrictions) => Links.Count(restrictions);
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
58     ↪ => Links.Each(handler, restrictions);
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     public virtual TLink Create(IList<TLink> restrictions) => Links.Create(restrictions);
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
65     ↪ Links.Update(restrictions, substitution);
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);
69
70     protected override bool AllowMultipleDisposeCalls
71     {
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         get => true;
74     }
75
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     protected override void Dispose(bool manual, bool wasDisposed)
78     {
79         if (!wasDisposed)
80         {
81             Links.DisposeIfPossible();
82         }
83     }
84 }

```

1.5 ./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             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
20             return Links.Each(handler, restrictions);
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
25         {
26             // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
27             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
28             Links.EnsureInnerReferenceExists(substitution, nameof(substitution));
29             return Links.Update(restrictions, substitution);
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public override void Delete(IList<TLink> restrictions)
34         {

```

```

34         var link = restrictions[Constants.IndexPart];
35         Links.EnsureLinkExists(link, nameof(link));
36         Links.Delete(link);
37     }
38 }
39 }

```

1.6 ./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             var indexPartConstant = constants.IndexPart;
23             var sourcePartConstant = constants.SourcePart;
24             var targetPartConstant = constants.TargetPart;
25             var restrictionsCount = restrictions.Count;
26             if (!_equalityComparer.Equals(constants.Any, itselfConstant)
27                 && (((restrictionsCount > indexPartConstant) &&
28                     ⇨ _equalityComparer.Equals(restrictions[indexPartConstant], itselfConstant))
29                 || ((restrictionsCount > sourcePartConstant) &&
30                     ⇨ _equalityComparer.Equals(restrictions[sourcePartConstant], itselfConstant))
31                 || ((restrictionsCount > targetPartConstant) &&
32                     ⇨ _equalityComparer.Equals(restrictions[targetPartConstant], itselfConstant))))
33             {
34                 // Itself constant is not supported for Each method right now, skipping execution
35                 return constants.Continue;
36             }
37             return Links.Each(handler, restrictions);
38         }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
42             ⇨ Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Itself,
43                 ⇨ restrictions, substitution));
44     }
45 }

```

1.7 ./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             Links.EnsureCreated(substitution[constants.SourcePart],
25                 ⇨ substitution[constants.TargetPart]);
26         }
27     }
28 }

```

```

23         return Links.Update(restrictions, substitution);
24     }
25 }
26 }

```

1.8 ./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 ./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 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
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
32             ↪ newLinkAddress)
33         {
34             if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
35             ↪ Links.Exists(oldLinkAddress))
36             {
37                 Facade.Delete(oldLinkAddress);
38             }
39             return newLinkAddress;
40         }
41     }
42 }

```

1.10 ./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             Links.EnsureDoesNotExists(substitution[Constants.SourcePart],
17                                     ↪ substitution[Constants.TargetPart]);
18             return Links.Update(restrictions, substitution);
19         }
20     }

```

1.11 ./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             Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
17             return Links.Update(restrictions, substitution);
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public override void Delete(IList<TLink> restrictions)
22         {
23             var link = restrictions[Constants.IndexPart];
24             Links.EnsureNoUsages(link);
25             Links.Delete(link);
26         }
27     }
28 }

```

1.12 ./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             Links.EnforceResetValues(linkIndex);
18             Links.Delete(linkIndex);
19         }
20     }
21 }

```

1.13 ./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      /// Представляет объект для работы с базой данных (файлом) в формате Links (массива связей).
10     /// </summary>
11     /// <remarks>
12     /// Возможные оптимизации:
13     /// Объединение в одно поле Source и Target с уменьшением до 32 бит.
14     ///     + меньше объём БД
15     ///     - меньше производительность
16     ///     - больше ограничение на количество связей в БД)
17     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
18     ///     + меньше объём БД
19     ///     - больше сложность
20     ///
21     /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
22     ↪ поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
23     ↪ 460 752 303 423 488
24     /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
25     ↪ (битовыми строками) - вариант матрицы (выстраиваемой лениво).
26     ///
27     /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
28     ↪ выбрасываться только при #if DEBUG
29     /// </remarks>
30     public class UInt64Links : LinksDisposableDecoratorBase<ulong>
31     {
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public UInt64Links(ILinks<ulong> links) : base(links) { }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public override ulong Create(IList<ulong> restrictions) => Links.CreatePoint();
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
40         {
41             var constants = Constants;
42             var indexPartConstant = constants.IndexPart;
43             var sourcePartConstant = constants.SourcePart;
44             var targetPartConstant = constants.TargetPart;
45             var nullConstant = constants.Null;
46             var itselfConstant = constants.Itself;
47             var existedLink = nullConstant;
48             var updatedLink = restrictions[indexPartConstant];
49             var newSource = substitution[sourcePartConstant];
50             var newTarget = substitution[targetPartConstant];
51             if (newSource != itselfConstant && newTarget != itselfConstant)
52             {
53                 existedLink = Links.SearchOrDefault(newSource, newTarget);
54             }
55             if (existedLink == nullConstant)
56             {
57                 var before = Links.GetLink(updatedLink);
58                 if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
59                     ↪ newTarget)
60                 {
61                     Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
62                         ↪ newSource,
63                                     newTarget == itselfConstant ? updatedLink :
64                                         ↪ newTarget);
65                 }
66                 return updatedLink;
67             }
68             else
69             {
70                 return Facade.MergeAndDelete(updatedLink, existedLink);
71             }
72         }
73
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         public override void Delete(IList<ulong> restrictions)
76         {
77             var linkIndex = restrictions[Constants.IndexPart];
78             Links.EnforceResetValues(linkIndex);
79             Facade.DeleteAllUsages(linkIndex);
80             Links.Delete(linkIndex);
81         }
82     }
83 }

```

1.14 ./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         //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
41         //public static readonly IReadOnlyList<TLink> NullLink = new
42         //    ↳ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
43         //    ↳ });
44
45         // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
46         //    ↳ (Links-Expression)
47         public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
48             ↳ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
49             ↳ substitutedHandler)
50         {
51             /////List<Transition> transitions = null;
52             /////if (!restriction.IsNullOrEmpty())
53             /////{
54             /////    // Есть причина делать проход (чтение)
55             /////    if (matchedHandler != null)
56             /////    {
57             /////        if (!substitution.IsNullOrEmpty())
58             /////        {
59             /////            // restriction => { 0, 0, 0 } | { 0 } // Create
60             /////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
61             /////            ↳ Create / Update
62             /////            // substitution => { 0, 0, 0 } | { 0 } // Delete
63             /////            transitions = new List<Transition>();
64             /////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
65             /////            {
66             /////                // If index is Null, that means we always ignore every other
67             /////                ↳ value (they are also Null by definition)
68             /////                var matchDecision = matchedHandler(, NullLink);
69             /////                if (Equals(matchDecision, Constants.Break))
70             /////                    return false;
71             /////                if (!Equals(matchDecision, Constants.Skip))
72             /////                    transitions.Add(new Transition(matchedLink, newValue));
73             /////            }
74             /////            else
75             /////            {
76             /////                Func<T, bool> handler;
77             /////                handler = link =>
78             /////                {
79             /////                    var matchedLink = Memory.GetLinkValue(link);
80             /////                    var newValue = Memory.GetLinkValue(link);
81             /////                    newValue[Constants.IndexPart] = Constants.Itself;
82             /////                }
83             /////            }
84             /////        }
85             /////    }
86         }
87     }
88 }

```



```

71      newValue[Constants.SourcePart] =
    ↳ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
    ↳ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
72      newValue[Constants.TargetPart] =
    ↳ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
    ↳ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
73      var matchDecision = matchedHandler(matchedLink, newValue);
74      if (Equals(matchDecision, Constants.Break))
75          return false;
76      if (!Equals(matchDecision, Constants.Skip))
77          transitions.Add(new Transition(matchedLink, newValue));
78      return true;
79  };
80  if (!Memory.Each(handler, restriction))
81      return Constants.Break;
82  }
83  }
84  else
85  {
86      Func<T, bool> handler = link =>
87      {
88          var matchedLink = Memory.GetLinkValue(link);
89          var matchDecision = matchedHandler(matchedLink, matchedLink);
90          return !Equals(matchDecision, Constants.Break);
91      };
92      if (!Memory.Each(handler, restriction))
93          return Constants.Break;
94  }
95  }
96  else
97  {
98      if (substitution != null)
99      {
100         transitions = new List<IList<T>>();
101         Func<T, bool> handler = link =>
102         {
103             var matchedLink = Memory.GetLinkValue(link);
104             transitions.Add(matchedLink);
105             return true;
106         };
107         if (!Memory.Each(handler, restriction))
108             return Constants.Break;
109     }
110     else
111     {
112         return Constants.Continue;
113     }
114 }
115 }
116 if (substitution != null)
117 {
118     // Есть причина делать замену (запись)
119     if (substitutedHandler != null)
120     {
121     }
122     else
123     {
124     }
125 }
126 return Constants.Continue;
127
128 //if (restriction.IsNullOrEmpty()) // Create
129 //{
130 //    substitution[Constants.IndexPart] = Memory.AllocateLink();
131 //    Memory.SetLinkValue(substitution);
132 //}
133 //else if (restriction.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)),
    ↳ Constants.Break), restriction))

```

```

143         //         return Constants.Break;
144     //}
145     //else // Update
146     //{
147         //     //List<IList<T>> matchedLinks = null;
148         //     if (matchedHandler != null)
149         //     {
150             //         matchedLinks = new List<IList<T>>();
151             //         Func<T, bool> handler = link =>
152             //         {
153                 //             var matchedLink = Memory.GetLinkValue(link);
154                 //             var matchDecision = matchedHandler(matchedLink);
155                 //             if (Equals(matchDecision, Constants.Break))
156                     //                 return false;
157                 //             if (!Equals(matchDecision, Constants.Skip))
158                     //                 matchedLinks.Add(matchedLink);
159                 //             return true;
160             //         };
161             //         if (!Memory.Each(handler, restriction))
162                 //             return Constants.Break;
163             //     }
164             //     if (!matchedLinks.IsNullOrEmpty())
165             //     {
166                 //         var totalMatchedLinks = matchedLinks.Count;
167                 //         for (var i = 0; i < totalMatchedLinks; i++)
168                 //         {
169                     //             var matchedLink = matchedLinks[i];
170                     //             if (substitutedHandler != null)
171                     //             {
172                         //                 var newValue = new List<T>(); // TODO: Prepare value to update here
173                         //                 // TODO: Decide is it actually needed to use Before and After
174                         //                 substitution handling.
175                         //                 var substitutedDecision = substitutedHandler(matchedLink,
176                         //                 newValue);
177                         //                 if (Equals(substitutedDecision, Constants.Break))
178                             //                     return Constants.Break;
179                         //                 if (Equals(substitutedDecision, Constants.Continue))
180                         //                 {
181                             //                     // Actual update here
182                             //                     Memory.SetLinkValue(newValue);
183                             //                 }
184                         //                 if (Equals(substitutedDecision, Constants.Skip))
185                             //                 {
186                             //                     // Cancel the update. TODO: decide use separate Cancel
187                             //                     // constant or Skip is enough?
188                             //                 }
189                         //             }
190                     //         }
191                 //     }
192             // }
193         // }
194     //}
195     return Constants.Continue;
196 }
197
198 public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
199     ↳ matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
200     ↳ substitutionHandler)
201 {
202     if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
203     {
204         return Constants.Continue;
205     }
206     else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
207         ↳ Check if it is a correct condition
208     {
209         // Or it only applies to trigger without matchHandler.
210         throw new NotImplementedException();
211     }
212     else if (!substitution.IsNullOrEmpty()) // Creation
213     {
214         var before = Array.Empty<TLink>();
215         // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
216         ↳ (пройти мимо) или пустить (взять)?
217         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
218         ↳ Constants.Break))
219         {
220             return Constants.Break;
221         }
222         var after = (IList<TLink>)substitution.ToArray();

```

```

213     if (_equalityComparer.Equals(after[0], default))
214     {
215         var newLink = Links.Create();
216         after[0] = newLink;
217     }
218     if (substitution.Count == 1)
219     {
220         after = Links.GetLink(substitution[0]);
221     }
222     else if (substitution.Count == 3)
223     {
224         //Links.Create(after);
225     }
226     else
227     {
228         throw new NotSupportedException();
229     }
230     if (matchHandler != null)
231     {
232         return substitutionHandler(before, after);
233     }
234     return Constants.Continue;
235 }
236 else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
237 {
238     if (patternOrCondition.Count == 1)
239     {
240         var linkToDelete = patternOrCondition[0];
241         var before = Links.GetLink(linkToDelete);
242         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
243             ↪ Constants.Break))
244         {
245             return Constants.Break;
246         }
247         var after = Array.Empty<TLink>();
248         Links.Update(linkToDelete, Constants.Null, Constants.Null);
249         Links.Delete(linkToDelete);
250         if (matchHandler != null)
251         {
252             return substitutionHandler(before, after);
253         }
254         return Constants.Continue;
255     }
256     else
257     {
258         throw new NotSupportedException();
259     }
260 }
261 else // Replace / Update
262 {
263     if (patternOrCondition.Count == 1) //-V3125
264     {
265         var linkToUpdate = patternOrCondition[0];
266         var before = Links.GetLink(linkToUpdate);
267         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
268             ↪ Constants.Break))
269         {
270             return Constants.Break;
271         }
272         var after = (IList<TLink>)substitution.ToArray(); //-V3125
273         if (_equalityComparer.Equals(after[0], default))
274         {
275             after[0] = linkToUpdate;
276         }
277         if (substitution.Count == 1)
278         {
279             if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
280             {
281                 after = Links.GetLink(substitution[0]);
282                 Links.Update(linkToUpdate, Constants.Null, Constants.Null);
283                 Links.Delete(linkToUpdate);
284             }
285         }
286         else if (substitution.Count == 3)
287         {
288             //Links.Update(after);
289         }
290     }
291     else

```

```

289         {
290             throw new NotSupportedException();
291         }
292         if (matchHandler != null)
293         {
294             return substitutionHandler(before, after);
295         }
296         return Constants.Continue;
297     }
298     else
299     {
300         throw new NotSupportedException();
301     }
302 }
303
304
305 /// <remarks>
306 /// IList[IList[IList[T]]]
307 /// |         |         |         |
308 /// |         |         |         |
309 /// |         |         |         |
310 /// |         |         |         |
311 /// |         |         |         |
312 /// |         |         |         |
313 /// |         |         |         |
314 /// |         |         |         |
315 public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
    ↳ substitution)
316 {
317     var changes = new List<IList<IList<TLink>>>();
318     Trigger(condition, AlwaysContinue, substitution, (before, after) =>
319     {
320         var change = new[] { before, after };
321         changes.Add(change);
322         return Constants.Continue;
323     });
324     return changes;
325 }
326
327 private TLink AlwaysContinue(IList<TLink> linkToMatch) => Constants.Continue;
328 }
329 }

```

1.15 ./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)]

```

```

36     public override string ToString() => $"{Source}->{Target}";
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
40     ↪ && _equalityComparer.Equals(Target, other.Target);
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public override bool Equals(object obj) => obj is Doublet<T> doublet ?
44     ↪ base.Equals(doublet) : false;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     public override int GetHashCode() => (Source, Target).GetHashCode();
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     public static bool operator !=(Doublet<T> left, Doublet<T> right) => !(left == right);
54 }
55 }

```

1.16 ./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 ./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 ./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
    ↳ amountOfCreations)
23 {
24     var random = RandomHelpers.Default;
25     var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
26     var uint64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
27     for (var i = OUL; i < amountOfCreations; i++)
28     {
29         var linksAddressRange = new Range<ulong>(0,
    ↳ addressToUInt64Converter.Convert(links.Count()));
30         var source =
    ↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
31         var target =
    ↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
32         links.GetOrCreate(source, target);
33     }
34 }
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 public static void RunRandomSearches<TLink>(this ILinks<TLink> links, ulong
    ↳ amountOfSearches)
38 {
39     var random = RandomHelpers.Default;
40     var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
41     var uint64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
42     for (var i = OUL; i < amountOfSearches; i++)
43     {
44         var linksAddressRange = new Range<ulong>(0,
    ↳ addressToUInt64Converter.Convert(links.Count()));
45         var source =
    ↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
46         var target =
    ↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
47         links.SearchOrDefault(source, target);
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
113             ↳ найдено связей.");
114     }
115     return firstLink;
116 }
117
118 #region Paths
119
120 /// <remarks>
121 /// TODO: Как так? Как то что ниже может быть корректно?
122 /// Скорее всего практически не применимо
123 /// Предполагалось, что можно было конвертировать формируемый в проходе через
124 /// ↳ SequenceWalker
125 /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
126 /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
127 /// </remarks>
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
130     ↳ path)
131 {
132     var current = path[0];
133     //EnsureLinkExists(current, "path");
134     if (!links.Exists(current))
135     {
136         return false;
137     }
138     var equalityComparer = EqualityComparer<TLink>.Default;
139     var constants = links.Constants;
140     for (var i = 1; i < path.Length; i++)
141     {
142         var next = path[i];
143         var values = links.GetLink(current);
144         var source = values[constants.SourcePart];
145         var target = values[constants.TargetPart];
146         if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
147             ↳ next))
148         {
149             //throw new InvalidOperationException(string.Format("Невозможно выбрать
150             ↳ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
151             return false;
152         }
153         if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
154             ↳ target))
155         {
156             //throw new InvalidOperationException(string.Format("Невозможно продолжить
157             ↳ путь через элемент пути {0}", next));
158             return false;
159         }
160         current = next;
161     }
162     return true;
163 }
164
165 /// <remarks>

```

```

159  /// Может потребовать дополнительного стека для PathElement's при использовании
160  ↪ SequenceWalker.
161  /// </remarks>
162  [MethodImpl(MethodImplOptions.AggressiveInlining)]
163  public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
164  ↪ path)
165  {
166      links.EnsureLinkExists(root, "root");
167      var currentLink = root;
168      for (var i = 0; i < path.Length; i++)
169      {
170          currentLink = links.GetLink(currentLink)[path[i]];
171      }
172      return currentLink;
173  }
174  [MethodImpl(MethodImplOptions.AggressiveInlining)]
175  public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
176  ↪ links, TLink root, ulong size, ulong index)
177  {
178      var constants = links.Constants;
179      var source = constants.SourcePart;
180      var target = constants.TargetPart;
181      if (!Platform.Numbers.Math.IsPowerOfTwo(size))
182      {
183          throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
184          ↪ than powers of two are not supported.");
185      }
186      var path = new BitArray(BitConverter.GetBytes(index));
187      var length = Bit.GetLowestPosition(size);
188      links.EnsureLinkExists(root, "root");
189      var currentLink = root;
190      for (var i = length - 1; i >= 0; i--)
191      {
192          currentLink = links.GetLink(currentLink)[path[i] ? target : source];
193      }
194      return currentLink;
195  }
196  #endregion
197  /// <summary>
198  /// Возвращает индекс указанной связи.
199  /// </summary>
200  /// <param name="links">Хранилище связей.</param>
201  /// <param name="link">Связь представленная списком, состоящим из её адреса и
202  ↪ содержимого.</param>
203  /// <returns>Индекс начальной связи для указанной связи.</returns>
204  [MethodImpl(MethodImplOptions.AggressiveInlining)]
205  public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
206  ↪ link[links.Constants.IndexPart];
207  /// <summary>
208  /// Возвращает индекс начальной (Source) связи для указанной связи.
209  /// </summary>
210  /// <param name="links">Хранилище связей.</param>
211  /// <param name="link">Индекс связи.</param>
212  /// <returns>Индекс начальной связи для указанной связи.</returns>
213  [MethodImpl(MethodImplOptions.AggressiveInlining)]
214  public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
215  ↪ links.GetLink(link)[links.Constants.SourcePart];
216  /// <summary>
217  /// Возвращает индекс начальной (Source) связи для указанной связи.
218  /// </summary>
219  /// <param name="links">Хранилище связей.</param>
220  /// <param name="link">Связь представленная списком, состоящим из её адреса и
221  ↪ содержимого.</param>
222  /// <returns>Индекс начальной связи для указанной связи.</returns>
223  [MethodImpl(MethodImplOptions.AggressiveInlining)]
224  public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
225  ↪ link[links.Constants.SourcePart];
226  /// <summary>
227  /// Возвращает индекс конечной (Target) связи для указанной связи.
228  /// </summary>
229  /// <param name="links">Хранилище связей.</param>
230  /// <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)
250     => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
    ↳ links.Constants.Continue);
251
252 /// <summary>
253 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
254 /// </summary>
255 /// <param name="links">Хранилище связей.</param>
256 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
    ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
257 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
    ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
258 /// <param name="handler">Обработчик каждой подходящей связи.</param>
259 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
260 [MethodImpl(MethodImplOptions.AggressiveInlining)]
261 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<TLink, bool> handler)
262 {
263     var constants = links.Constants;
264     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
    ↳ constants.Break, constants.Any, source, target);
265 }
266
267 /// <summary>
268 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
269 /// </summary>
270 /// <param name="links">Хранилище связей.</param>
271 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
    ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
272 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
    ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
273 /// <param name="handler">Обработчик каждой подходящей связи.</param>
274 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
275 [MethodImpl(MethodImplOptions.AggressiveInlining)]
276 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<IList<TLink>, TLink> handler) => links.Each(handler, links.Constants.Any,
    ↳ source, target);
277
278 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

279 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ restrictions)
280 {
281     var arraySize = CheckedConverter<TLink,
    ↳ long>.Default.Convert(links.Count(restrictions));
282     if (arraySize > 0)
283     {
284         var array = new IList<TLink>[arraySize];
285         var filler = new ArrayFiller<IList<TLink>, TLink>(array,
    ↳ links.Constants.Continue);
286         links.Each(filler.AddAndReturnConstant, restrictions);
287         return array;
288     }
289     else
290     {
291         return Array.Empty<IList<TLink>>();
292     }
293 }
294
295 [MethodImpl(MethodImplOptions.AggressiveInlining)]
296 public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ restrictions)
297 {
298     var arraySize = CheckedConverter<TLink,
    ↳ long>.Default.Convert(links.Count(restrictions));
299     if (arraySize > 0)
300     {
301         var array = new TLink[arraySize];
302         var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
303         links.Each(filler.AddFirstAndReturnConstant, restrictions);
304         return array;
305     }
306     else
307     {
308         return Array.Empty<TLink>();
309     }
310 }
311
312 /// <summary>
313 /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
    ↳ в хранилище связей.
314 /// </summary>
315 /// <param name="links">Хранилище связей.</param>
316 /// <param name="source">Начало связи.</param>
317 /// <param name="target">Конец связи.</param>
318 /// <returns>Значение, определяющее существует ли связь.</returns>
319 [MethodImpl(MethodImplOptions.AggressiveInlining)]
320 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
    ↳ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
    ↳ default) > 0;
321
322 #region Ensure
323 // TODO: May be move to EnsureExtensions or make it both there and here
324
325 [MethodImpl(MethodImplOptions.AggressiveInlining)]
326 public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↳ restrictions)
327 {
328     for (var i = 0; i < restrictions.Count; i++)
329     {
330         if (!links.Exists(restrictions[i]))
331         {
332             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
    ↳ $"sequence[{i}]");
333         }
334     }
335 }
336
337 [MethodImpl(MethodImplOptions.AggressiveInlining)]
338 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
    ↳ reference, string argumentName)
339 {
340     if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
341     {
342         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
343     }
344 }
345

```

```

346 [MethodImpl(MethodImplOptions.AggressiveInlining)]
347 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
    ↳ IList<TLink> restrictions, string argumentName)
348 {
349     for (int i = 0; i < restrictions.Count; i++)
350     {
351         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
352     }
353 }
354
355 [MethodImpl(MethodImplOptions.AggressiveInlining)]
356 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↳ restrictions)
357 {
358     var equalityComparer = EqualityComparer<TLink>.Default;
359     var any = links.Constants.Any;
360     for (var i = 0; i < restrictions.Count; i++)
361     {
362         if (!equalityComparer.Equals(restrictions[i], any) &&
            ↳ !links.Exists(restrictions[i]))
363         {
364             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                ↳ $"sequence[{i}]");
365         }
366     }
367 }
368
369 [MethodImpl(MethodImplOptions.AggressiveInlining)]
370 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
    ↳ string argumentName)
371 {
372     var equalityComparer = EqualityComparer<TLink>.Default;
373     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
374     {
375         throw new ArgumentLinkDoesNotExistsException<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 ArgumentLinkDoesNotExistsException<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 ArgumentLinkHasDependenciesException<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
445 #endregion
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,
        ↳ constants.Any));
454     var equalityComparer = EqualityComparer<TLink>.Default;
455     if (equalityComparer.Equals(values[constants.SourcePart], link))
456     {
457         usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
458     }
459     TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
        ↳ link));
460     if (equalityComparer.Equals(values[constants.TargetPart], link))
461     {
462         usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
463     }
464     return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
465 }
466
467 /// <param name="links">Хранилище связей.</param>
468 [MethodImpl(MethodImplOptions.AggressiveInlining)]
469 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
    ↳ Comparer<TLink>.Default.Compare(links.CountUsages(link), default) > 0;
470
471 /// <param name="links">Хранилище связей.</param>
472 [MethodImpl(MethodImplOptions.AggressiveInlining)]
473 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
    ↳ TLink target)
474 {
475     var constants = links.Constants;
476     var values = links.GetLink(link);
477     var equalityComparer = EqualityComparer<TLink>.Default;
478     return equalityComparer.Equals(values[constants.SourcePart], source) &&
        ↳ equalityComparer.Equals(values[constants.TargetPart], target);
479 }
480
481 /// <summary>

```

```

482    /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
483    /// </summary>
484    /// <param name="links">Хранилище связей.</param>
485    /// <param name="source">Индекс связи, которая является началом для искомой
    → связи.</param>
486    /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
487    /// <returns>Индекс искомой связи с указанными Source (началом) и Target
    → (концом).</returns>
488    [MethodImpl(MethodImplOptions.AggressiveInlining)]
489    public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
    → target)
490    {
491        var constants = links.Constants;
492        var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
493        links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
494        return setter.Result;
495    }
496
497    /// <param name="links">Хранилище связей.</param>
498    [MethodImpl(MethodImplOptions.AggressiveInlining)]
499    public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
500
501    /// <param name="links">Хранилище связей.</param>
502    [MethodImpl(MethodImplOptions.AggressiveInlining)]
503    public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
504    {
505        var link = links.Create();
506        return links.Update(link, link, link);
507    }
508
509    /// <param name="links">Хранилище связей.</param>
510    [MethodImpl(MethodImplOptions.AggressiveInlining)]
511    public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
    → target) => links.Update(links.Create(), source, target);
512
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
525    /// <summary>
526    /// Обновляет связь с указанными началом (Source) и концом (Target)
527    /// на связь с указанными началом (NewSource) и концом (NewTarget).
528    /// </summary>
529    /// <param name="links">Хранилище связей.</param>
530    /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
    → может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
    → Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
    → связи.</param>
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    }

```

```

547 }
548
549 [MethodImpl(MethodImplOptions.AggressiveInlining)]
550 public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
    ↳ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
551 {
552     var equalityComparer = EqualityComparer<TLink>.Default;
553     var constants = links.Constants;
554     var restrictionsIndex = restrictions[constants.IndexPart];
555     var substitutionIndex = substitution[constants.IndexPart];
556     if (equalityComparer.Equals(substitutionIndex, default))
557     {
558         substitutionIndex = restrictionsIndex;
559     }
560     var source = substitution[constants.SourcePart];
561     var target = substitution[constants.TargetPart];
562     source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
563     target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
564     return new Link<TLink>(substitutionIndex, source, target);
565 }
566
567 /// <summary>
568 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
    ↳ с указанными Source (началом) и Target (концом).
569 /// </summary>
570 /// <param name="links">Хранилище связей.</param>
571 /// <param name="source">Индекс связи, которая является началом на создаваемой
    ↳ связи.</param>
572 /// <param name="target">Индекс связи, которая является концом для создаваемой
    ↳ связи.</param>
573 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
574 [MethodImpl(MethodImplOptions.AggressiveInlining)]
575 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)
576 {
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,
            ↪ links.Constants.Continue);
658         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
659         for (var i = count - 1; i >= 0; i--)
660         {
661             links.Delete(queryResult[i]);
662         }
663     }
664 }
665
666 // TODO: Move to Platform.Data
667 [MethodImpl(MethodImplOptions.AggressiveInlining)]
668 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
669 {
670     var nullConstant = links.Constants.Null;
671     var equalityComparer = EqualityComparer<TLink>.Default;
672     var link = links.GetLink(linkIndex);
673     for (int i = 1; i < link.Count; i++)
674     {
675         if (!equalityComparer.Equals(link[i], nullConstant))
676         {
677             return false;
678         }
679     }
680     return true;
681 }
682
683 // TODO: Create a universal version of this method in Platform.Data (with using of for
    ↪ loop)
684 [MethodImpl(MethodImplOptions.AggressiveInlining)]
685 public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
686 {

```

```

687     var nullConstant = links.Constants.Null;
688     var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
689     links.Update(updateRequest);
690 }
691
692 // TODO: Create a universal version of this method in Platform.Data (with using of for
693     ↳ loop)
694 [MethodImpl(MethodImplOptions.AggressiveInlining)]
695 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
696 {
697     if (!links.AreValuesReset(linkIndex))
698     {
699         links.ResetValues(linkIndex);
700     }
701 }
702
703 /// <summary>
704 /// Merging two usages graphs, all children of old link moved to be children of new link
705     ↳ or deleted.
706 /// </summary>
707 [MethodImpl(MethodImplOptions.AggressiveInlining)]
708 public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
709     ↳ TLink newLinkIndex)
710 {
711     var addressToInt64Converter = CheckedConverter<TLink, long>.Default;
712     var equalityComparer = EqualityComparer<TLink>.Default;
713     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
714     {
715         var constants = links.Constants;
716         var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
717             ↳ constants.Any);
718         var usagesAsSourceCount =
719             ↳ addressToInt64Converter.Convert(links.Count(usagesAsSourceQuery));
720         var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
721             ↳ oldLinkIndex);
722         var usagesAsTargetCount =
723             ↳ addressToInt64Converter.Convert(links.Count(usagesAsTargetQuery));
724         var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
725             ↳ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
726         if (!isStandalonePoint)
727         {
728             var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
729             if (totalUsages > 0)
730             {
731                 var usages = ArrayPool.Allocate<TLink>(totalUsages);
732                 var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
733                     ↳ links.Constants.Continue);
734                 var i = 0L;
735                 if (usagesAsSourceCount > 0)
736                 {
737                     links.Each(usagesFiller.AddFirstAndReturnConstant,
738                         ↳ usagesAsSourceQuery);
739                     for (; i < usagesAsSourceCount; i++)
740                     {
741                         var usage = usages[i];
742                         if (!equalityComparer.Equals(usage, oldLinkIndex))
743                         {
744                             links.Update(usage, newLinkIndex, links.GetTarget(usage));
745                         }
746                     }
747                 }
748                 if (usagesAsTargetCount > 0)
749                 {
750                     links.Each(usagesFiller.AddFirstAndReturnConstant,
751                         ↳ usagesAsTargetQuery);
752                     for (; i < usages.Length; i++)
753                     {
754                         var usage = usages[i];
755                         if (!equalityComparer.Equals(usage, oldLinkIndex))
756                         {
757                             links.Update(usage, links.GetSource(usage), newLinkIndex);
758                         }
759                     }
760                 }
761                 ArrayPool.Free(usages);
762             }
763         }
764     }

```



```

753     }
754     return newLinkIndex;
755 }
756
757 /// <summary>
758 /// Replace one link with another (replaced link is deleted, children are updated or
759   ↳ deleted).
760 /// </summary>
761 [MethodImpl(MethodImplOptions.AggressiveInlining)]
762 public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
763   ↳ TLink newLinkIndex)
764 {
765     var equalityComparer = EqualityComparer<TLink>.Default;
766     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
767     {
768         links.MergeUsages(oldLinkIndex, newLinkIndex);
769         links.Delete(oldLinkIndex);
770     }
771     return newLinkIndex;
772 }
773
774 [MethodImpl(MethodImplOptions.AggressiveInlining)]
775 public static ILinks<TLink>
776   ↳ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
777 {
778     links = new LinksCascadeUsagesResolver<TLink>(links);
779     links = new NonNullContentsLinkDeletionResolver<TLink>(links);
780     links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
781     return links;
782 }

```

1.19 ./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>,
6         ↳ LinksConstants<TLink>>, ILinks<TLink>
7      {
8      }
9  }

```

1.20 ./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             if (_equalityComparer.Equals(frequency, default))
32             {
33                 return Links.GetOrCreate(_unaryOne, _frequencyMarker);
34             }
35         }
36     }

```

```

33         var incrementedSource =
34             ↪ _unaryNumberIncrementer.Increment(Links.GetSource(frequency));
35         return Links.GetOrCreate(incrementedSource, _frequencyMarker);
36     }
37 }

```

1.21 ./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            if (_equalityComparer.Equals(unaryNumber, _unaryOne))
24            {
25                return Links.GetOrCreate(_unaryOne, _unaryOne);
26            }
27            var source = Links.GetSource(unaryNumber);
28            var target = Links.GetTarget(unaryNumber);
29            if (_equalityComparer.Equals(source, target))
30            {
31                return Links.GetOrCreate(unaryNumber, _unaryOne);
32            }
33            else
34            {
35                return Links.GetOrCreate(source, Increment(target));
36            }
37        }
38    }
39 }

```

1.22 ./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 }

```

```

30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
    ↪ Target);
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 public Link(ICollection<TLink> values) => SetValues(values, out Index, out Source, out Target);
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 public Link(object other)
38 {
39     if (other is Link<TLink> otherLink)
40     {
41         SetValues(ref otherLink, out Index, out Source, out Target);
42     }
43     else if (other is ICollection<TLink> otherList)
44     {
45         SetValues(otherList, out Index, out Source, out Target);
46     }
47     else
48     {
49         throw new NotSupportedException();
50     }
51 }
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
    ↪ Target);
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 public Link(TLink index, TLink source, TLink target)
58 {
59     Index = index;
60     Source = source;
61     Target = target;
62 }
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
    ↪ out TLink target)
66 {
67     index = other.Index;
68     source = other.Source;
69     target = other.Target;
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 private static void SetValues(ICollection<TLink> values, out TLink index, out TLink source,
    ↪ out TLink target)
74 {
75     switch (values.Count)
76     {
77         case 3:
78             index = values[0];
79             source = values[1];
80             target = values[2];
81             break;
82         case 2:
83             index = values[0];
84             source = values[1];
85             target = default;
86             break;
87         case 1:
88             index = values[0];
89             source = default;
90             target = default;
91             break;
92         default:
93             index = default;
94             source = default;
95             target = default;
96             break;
97     }
98 }
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
    && _equalityComparer.Equals(Source, _constants.Null)
105

```

```

        && _equalityComparer.Equals(Target, _constants.Null);

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

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

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

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

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

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

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

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

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

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

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

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

```

```

179 [MethodImpl(MethodImplOptions.AggressiveInlining)]
180 public void Add(TLink item) => throw new NotSupportedException();
181
182 [MethodImpl(MethodImplOptions.AggressiveInlining)]
183 public void Clear() => throw new NotSupportedException();
184
185 [MethodImpl(MethodImplOptions.AggressiveInlining)]
186 public bool Contains(TLink item) => IndexOf(item) >= 0;
187
188 [MethodImpl(MethodImplOptions.AggressiveInlining)]
189 public void CopyTo(TLink[] array, int arrayIndex)
190 {
191     Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
192     Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
193         ↪ nameof(arrayIndex));
194     if (arrayIndex + Length > array.Length)
195     {
196         throw new InvalidOperationException();
197     }
198     array[arrayIndex++] = Index;
199     array[arrayIndex++] = Source;
200     array[arrayIndex] = Target;
201 }
202
203 [MethodImpl(MethodImplOptions.AggressiveInlining)]
204 public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
205
206 [MethodImpl(MethodImplOptions.AggressiveInlining)]
207 public int IndexOf(TLink item)
208 {
209     if (_equalityComparer.Equals(Index, item))
210     {
211         return _constants.IndexPart;
212     }
213     if (_equalityComparer.Equals(Source, item))
214     {
215         return _constants.SourcePart;
216     }
217     if (_equalityComparer.Equals(Target, item))
218     {
219         return _constants.TargetPart;
220     }
221     return -1;
222 }
223
224 [MethodImpl(MethodImplOptions.AggressiveInlining)]
225 public void Insert(int index, TLink item) => throw new NotSupportedException();
226
227 [MethodImpl(MethodImplOptions.AggressiveInlining)]
228 public void RemoveAt(int index) => throw new NotSupportedException();
229
230 [MethodImpl(MethodImplOptions.AggressiveInlining)]
231 public static bool operator ==(Link<TLink> left, Link<TLink> right) =>
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 ./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 }

```

```

14     }
15 }

```

1.24 ./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         public ILinks<TLink> Links
10         {
11             [MethodImpl(MethodImplOptions.AggressiveInlining)]
12             get;
13         }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected LinksOperatorBase(ILinks<TLink> links) => Links = links;
17     }
18 }

```

1.25 ./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>,
12         ↪ IConverter<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↪ EqualityComparer<TLink>.Default;
16         private static readonly TLink _zero = default;
17         private static readonly TLink _one = Arithmetic.Increment(_zero);
18
19         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
23             ↪ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
24             ↪ powerOf2ToUnaryNumberConverter;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(TLink number)
28         {
29             var nullConstant = Links.Constants.Null;
30             var target = nullConstant;
31             for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
32                 ↪ NumericType<TLink>.BitsSize; i++)
33             {
34                 if (_equalityComparer.Equals(Bit.And(number, _one), _one))
35                 {
36                     target = _equalityComparer.Equals(target, nullConstant)
37                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
38                         : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
39                 }
40                 number = Bit.ShiftRight(number, 1);
41             }
42             return target;
43         }
44     }
45 }

```

1.26 ./Platform.Data.Doublets/Numbers/Unary/LinkToltsFrequencyNumberConveter.cs

```

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

```

```

10 {
11     public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
        ↳ IConverter<Doublet<TLink>, TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
        ↳ EqualityComparer<TLink>.Default;
14
15         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
16         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public LinkToItsFrequencyNumberConveter(
20             ILinks<TLink> links,
21             IProperty<TLink, TLink> frequencyPropertyOperator,
22             IConverter<TLink> unaryNumberToAddressConverter)
23             : base(links)
24         {
25             _frequencyPropertyOperator = frequencyPropertyOperator;
26             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public TLink Convert(Doublet<TLink> doublet)
31         {
32             var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
33             if (_equalityComparer.Equals(link, default))
34             {
35                 throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
36             }
37             var frequency = _frequencyPropertyOperator.Get(link);
38             if (_equalityComparer.Equals(frequency, default))
39             {
40                 return default;
41             }
42             var frequencyNumber = Links.GetSource(frequency);
43             return _unaryNumberToAddressConverter.Convert(frequencyNumber);
44         }
45     }
46 }

```

1.27 ./Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs

```

1 using System.Collections.Generic;
2 using Platform.Exceptions;
3 using Platform.Ranges;
4 using Platform.Converters;
5 using System.Runtime.CompilerServices;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
        ↳ IConverter<int, TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
        ↳ EqualityComparer<TLink>.Default;
14
15         private readonly TLink[] _unaryNumberPowersOf2;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
19         {
20             _unaryNumberPowersOf2 = new TLink[64];
21             _unaryNumberPowersOf2[0] = one;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public TLink Convert(int power)
26         {
27             Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
        ↳ - 1), nameof(power));
28             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
29             {
30                 return _unaryNumberPowersOf2[power];
31             }
32             var previousPowerOf2 = Convert(power - 1);
33             var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
34             _unaryNumberPowersOf2[power] = powerOf2;
35             return powerOf2;
36         }
37     }
38 }

```

```
37     }
38 }
```

1.28 ./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             _unaryOne = unaryOne;
30             _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public TLink Convert(TLink unaryNumber)
35         {
36             if (_equalityComparer.Equals(unaryNumber, default))
37             {
38                 return default;
39             }
40             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
41             {
42                 return _one;
43             }
44             var source = Links.GetSource(unaryNumber);
45             var target = Links.GetTarget(unaryNumber);
46             if (_equalityComparer.Equals(source, target))
47             {
48                 return _unaryToUInt64[unaryNumber];
49             }
50             else
51             {
52                 var result = _unaryToUInt64[source];
53                 TLink lastValue;
54                 while (!_unaryToUInt64.TryGetValue(target, out lastValue))
55                 {
56                     source = Links.GetSource(target);
57                     result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
58                     target = Links.GetTarget(target);
59                 }
60                 result = Arithmetic<TLink>.Add(result, lastValue);
61                 return result;
62             }
63         }
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
67             ↳ links, TLink unaryOne)
68         {
69             var unaryToUInt64 = new Dictionary<TLink, TLink>
70             {
71                 { unaryOne, _one }
72             };
73             var unary = unaryOne;
74             var number = _one;
75             for (var i = 1; i < 64; i++)
76             {
77                 var result = _unaryToUInt64[unary];
78                 TLink lastValue;
79                 while (!_unaryToUInt64.TryGetValue(number, out lastValue))
80                 {
81                     unary = Links.GetSource(number);
82                     result = Arithmetic<TLink>.Add(result, _unaryToUInt64[unary]);
83                     number = Links.GetTarget(number);
84                 }
85                 result = Arithmetic<TLink>.Add(result, lastValue);
86                 _unaryToUInt64[number] = result;
87                 number = Links.GetTarget(number);
88             }
89             return _unaryToUInt64;
90         }
91     }
92 }
```



```

72         unary = links.GetOrCreate(unary, unary);
73         number = Double(number);
74         unaryToUInt64.Add(unary, number);
75     }
76     return unaryToUInt64;
77 }
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 private static TLink Double(TLink number) =>
    ↪ _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
81 }
82 }

```

1.29 ./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 ./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 objectProperty = Links.SearchOrDefault(@object, property);
21             if (_equalityComparer.Equals(objectProperty, default))
22             {
23                 return default;
24             }
25             var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
26             if (valueLink == null)
27             {
28                 return default;
29             }
30             return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public void SetValue(TLink @object, TLink property, TLink value)
35         {
36             var objectProperty = Links.GetOrCreate(@object, property);
37             Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
38             Links.GetOrCreate(objectProperty, value);
39         }
40     }
41 }

```

1.31 ./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 =
        ↪ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _propertyMarker;
14         private readonly TLink _propertyValueMarker;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
        ↪ propertyValueMarker) : base(links)
18         {
19             _propertyMarker = propertyMarker;
20             _propertyValueMarker = propertyValueMarker;
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public TLink Get(TLink link)
25         {
26             var property = Links.SearchOrDefault(link, _propertyMarker);
27             return GetValue(GetContainer(property));

```

```

28     }
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     private TLink GetContainer(TLink property)
32     {
33         var valueContainer = default(TLink);
34         if (_equalityComparer.Equals(property, default))
35         {
36             return valueContainer;
37         }
38         var links = Links;
39         var constants = links.Constants;
40         var countinueConstant = constants.Continue;
41         var breakConstant = constants.Break;
42         var anyConstant = constants.Any;
43         var query = new Link<TLink>(anyConstant, property, anyConstant);
44         links.Each(candidate =>
45         {
46             var candidateTarget = links.GetTarget(candidate);
47             var valueTarget = links.GetTarget(candidateTarget);
48             if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
49             {
50                 valueContainer = links.GetIndex(candidate);
51                 return breakConstant;
52             }
53             return countinueConstant;
54         }, query);
55         return valueContainer;
56     }
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
60         ? default : Links.GetTarget(container);
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     public void Set(TLink link, TLink value)
64     {
65         var links = Links;
66         var property = links.GetOrCreate(link, _propertyMarker);
67         var container = GetContainer(property);
68         if (_equalityComparer.Equals(container, default))
69         {
70             links.GetOrCreate(property, value);
71         }
72         else
73         {
74             links.Update(container, property, value);
75         }
76     }
77 }

```

1.32 ./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<int, TLink> _int32ToAddressConverter =
24             ↳ UncheckedConverter<int, TLink>.Default;
25
26         protected readonly TLink Break;
27         protected readonly TLink Continue;
28     }
29 }

```

```

23     protected readonly byte* Links;
24     protected readonly byte* Header;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
28     ↪ byte* header)
29     {
30         Links = links;
31         Header = header;
32         Break = constants.Break;
33         Continue = constants.Continue;
34     }
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected abstract TLink GetTreeRoot();
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected abstract TLink GetBasePartValue(TLink link);
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
44     ↪ rootSource, TLink rootTarget);
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
48     ↪ rootSource, TLink rootTarget);
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
52     ↪ AsRef<LinksHeader<TLink>>(Header);
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
56     ↪ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
57     ↪ _addressToInt64Converter.Convert(link)));
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
61     {
62         ref var link = ref GetLinkReference(linkIndex);
63         return new Link<TLink>(linkIndex, link.Source, link.Target);
64     }
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
68     {
69         ref var firstLink = ref GetLinkReference(first);
70         ref var secondLink = ref GetLinkReference(second);
71         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
72         ↪ secondLink.Source, secondLink.Target);
73     }
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
77     {
78         ref var firstLink = ref GetLinkReference(first);
79         ref var secondLink = ref GetLinkReference(second);
80         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
81         ↪ secondLink.Source, secondLink.Target);
82     }
83
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
86     ↪ -5);
87
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
90     ↪ Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
91
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     protected virtual bool GetLeftIsChildValue(TLink value)
94     {
95         unchecked
96         {
97             //return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(previousValue,
98             ↪ 4, 1));
99             return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
100         }
101     }

```

```

90     }
91
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
94     {
95         unchecked
96         {
97             var previousValue = storedValue;
98             var modified = Bit<TLink>.PartialWrite(previousValue,
99                 ↪ _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(previousValue,
110                 ↪ 3, 1));
111             return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
112         }
113     }
114
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
117     {
118         unchecked
119         {
120             var previousValue = storedValue;
121             var modified = Bit<TLink>.PartialWrite(previousValue,
122                 ↪ _boolToAddressConverter.Convert(value), 3, 1);
123             storedValue = modified;
124         }
125     }
126
127     [MethodImpl(MethodImplOptions.AggressiveInlining)]
128     protected bool IsChild(TLink parent, TLink possibleChild)
129     {
130         var parentSize = GetSize(parent);
131         var childSize = GetSizeOrZero(possibleChild);
132         return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
133     }
134
135     [MethodImpl(MethodImplOptions.AggressiveInlining)]
136     protected virtual sbyte GetBalanceValue(TLink storedValue)
137     {
138         unchecked
139         {
140             var value = _addressToInt32Converter.Convert(Bit<TLink>.PartialRead(storedValue,
141                 ↪ 0, 3));
142             value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
143                 ↪ end of sbyte
144             return (sbyte)value;
145         }
146     }
147
148     [MethodImpl(MethodImplOptions.AggressiveInlining)]
149     protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
150     {
151         unchecked
152         {
153             var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
154                 ↪ value & 3);
155             var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
156             storedValue = modified;
157         }
158     }
159
160     public TLink this[TLink index]
161     {
162         [MethodImpl(MethodImplOptions.AggressiveInlining)]
163         get
164         {
165             var root = GetTreeRoot();
166             if (GreaterOrEqualThan(index, GetSize(root)))
167             {
168                 return Zero;
169             }
170         }
171     }

```

```

163     }
164     while (!EqualToZero(root))
165     {
166         var left = GetLeftOrDefault(root);
167         var leftSize = GetSizeOrZero(left);
168         if (LessThan(index, leftSize))
169         {
170             root = left;
171             continue;
172         }
173         if (AreEqual(index, leftSize))
174         {
175             return root;
176         }
177         root = GetRightOrDefault(root);
178         index = Subtract(index, Increment(leftSize));
179     }
180     return Zero; // TODO: Impossible situation exception (only if tree structure
181                 ↪ broken)
182 }
183
184 /// <summary>
185 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
186 ↪ (концом).
187 /// </summary>
188 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
189 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
190 /// <returns>Индекс искомой связи.</returns>
191 [MethodImpl(MethodImplOptions.AggressiveInlining)]
192 public TLink Search(TLink source, TLink target)
193 {
194     var root = GetTreeRoot();
195     while (!EqualToZero(root))
196     {
197         ref var rootLink = ref GetLinkReference(root);
198         var rootSource = rootLink.Source;
199         var rootTarget = rootLink.Target;
200         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
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;

```

```

237     while (!EqualToZero(root))
238     {
239         var @base = GetBasePartValue(root);
240         if (GreaterOrEqualThan(@base, link))
241         {
242             root = GetLeftOrDefault(root);
243         }
244         else
245         {
246             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
247             root = GetRightOrDefault(root);
248         }
249     }
250     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
251 }
252
253 [MethodImpl(MethodImplOptions.AggressiveInlining)]
254 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
255 {
256     var root = GetTreeRoot();
257     if (EqualToZero(root))
258     {
259         return Continue;
260     }
261     TLink first = Zero, current = root;
262     while (!EqualToZero(current))
263     {
264         var @base = GetBasePartValue(current);
265         if (GreaterOrEqualThan(@base, link))
266         {
267             if (AreEqual(@base, link))
268             {
269                 first = current;
270             }
271             current = GetLeftOrDefault(current);
272         }
273         else
274         {
275             current = GetRightOrDefault(current);
276         }
277     }
278     if (!EqualToZero(first))
279     {
280         current = first;
281         while (true)
282         {
283             if (AreEqual(handler(GetLinkValues(current)), Break))
284             {
285                 return Break;
286             }
287             current = GetNext(current);
288             if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
289             {
290                 break;
291             }
292         }
293     }
294     return Continue;
295 }
296
297 [MethodImpl(MethodImplOptions.AggressiveInlining)]
298 protected override void PrintNodeValue(TLink node, StringBuilder sb)
299 {
300     ref var link = ref GetLinkReference(node);
301     sb.Append(' ');
302     sb.Append(link.Source);
303     sb.Append(' - ');
304     sb.Append(' > ');
305     sb.Append(link.Target);
306 }
307 }
308 }
309 }

```

1.33 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected abstract TLink GetTreeRoot();
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected abstract TLink GetBasePartValue(TLink link);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
42             ↳ rootSource, TLink rootTarget);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
46             ↳ rootSource, TLink rootTarget);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
50             ↳ AsRef<LinksHeader<TLink>>(Header);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
54             ↳ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
55             ↳ _addressToInt64Converter.Convert(link)));
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
59         {
60             ref var link = ref GetLinkReference(linkIndex);
61             return new Link<TLink>(linkIndex, link.Source, link.Target);
62         }
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
66         {
67             ref var firstLink = ref GetLinkReference(first);
68             ref var secondLink = ref GetLinkReference(second);
69             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
70             ↳ secondLink.Source, secondLink.Target);
71         }
72
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
75         {
76             ref var firstLink = ref GetLinkReference(first);
77             ref var secondLink = ref GetLinkReference(second);
78             return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
79             ↳ secondLink.Source, secondLink.Target);
80         }
81
82         public TLink this[TLink index]
83         {
84

```



```

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
99     ↪ broken)
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

```

```

148         {
149             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
150             root = GetLeftOrDefault(root);
151         }
152     }
153     root = GetTreeRoot();
154     var totalLeftIgnore = Zero;
155     while (!EqualToZero(root))
156     {
157         var @base = GetBasePartValue(root);
158         if (GreaterOrEqualThan(@base, link))
159         {
160             root = GetLeftOrDefault(root);
161         }
162         else
163         {
164             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
165             root = GetRightOrDefault(root);
166         }
167     }
168     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
169 }
170
171 [MethodImpl(MethodImplOptions.AggressiveInlining)]
172 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
173     ↳ EachUsageCore(@base, GetTreeRoot(), handler);
174
175 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
176 ↳ low-level MSIL stack.
177 [MethodImpl(MethodImplOptions.AggressiveInlining)]
178 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
179 {
180     var @continue = Continue;
181     if (EqualToZero(link))
182     {
183         return @continue;
184     }
185     var linkBasePart = GetBasePartValue(link);
186     var @break = Break;
187     if (GreaterThan(linkBasePart, @base))
188     {
189         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
190         {
191             return @break;
192         }
193     }
194     else if (LessThan(linkBasePart, @base))
195     {
196         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
197         {
198             return @break;
199         }
200     }
201     else //if (linkBasePart == @base)
202     {
203         if (AreEqual(handler(GetLinkValues(link)), @break))
204         {
205             return @break;
206         }
207         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
208         {
209             return @break;
210         }
211         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
212         {
213             return @break;
214         }
215     }
216     return @continue;
217 }
218
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 protected override void PrintNodeValue(TLink node, StringBuilder sb)
221 {
222     ref var link = ref GetLinkReference(node);
223     sb.Append(' ');
224     sb.Append(link.Source);
225     sb.Append('-');

```

```

225         sb.Append('>');
226         sb.Append(link.Target);
227     }
228 }
229 }

```

1.34 ./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 unsafe override ref TLink GetLeftReference(TLink node) => ref
16             ↳ GetLinkReference(node).LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected unsafe 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 SetLeft(TLink node, TLink left) =>
30             ↳ GetLinkReference(node).LeftAsSource = left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(TLink node, TLink right) =>
34             ↳ GetLinkReference(node).RightAsSource = right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override TLink GetSize(TLink node) =>
38             ↳ GetSizeValue(GetLinkReference(node).SizeAsSource);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
42             ↳ GetLinkReference(node).SizeAsSource, size);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override bool GetLeftIsChild(TLink node) =>
46             ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override void SetLeftIsChild(TLink node, bool value) =>
50             ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override bool GetRightIsChild(TLink node) =>
54             ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsSource);
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override void SetRightIsChild(TLink node, bool value) =>
58             ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override sbyte GetBalance(TLink node) =>
62             ↳ GetBalanceValue(GetLinkReference(node).SizeAsSource);
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
66             ↳ GetLinkReference(node).SizeAsSource, value);
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsSource;
70
71         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

58     protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
        ↪ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
        ↪ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override void ClearNode(TLink node)
68     {
69         ref var link = ref GetLinkReference(node);
70         link.LeftAsSource = Zero;
71         link.RightAsSource = Zero;
72         link.SizeAsSource = Zero;
73     }
74 }
75 }

```

1.35 ./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 unsafe override ref TLink GetLeftReference(TLink node) => ref
            ↪ GetLinkReference(node).LeftAsSource;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected unsafe 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,
47         ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
48         ↪ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override void ClearNode(TLink node)
52     {
53         ref var link = ref GetLinkReference(node);
54         link.LeftAsSource = Zero;
55         link.RightAsSource = Zero;
56         link.SizeAsSource = Zero;
57     }
58 }

```

1.36 ./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> :
8          ↪ LinksAvlBalancedTreeMethodsBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksTargetsAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
12             ↪ byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected unsafe override ref TLink GetLeftReference(TLink node) => ref
16             ↪ GetLinkReference(node).LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected unsafe 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) =>
38             ↪ GetSizeValue(GetLinkReference(node).SizeAsTarget);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
42             ↪ GetLinkReference(node).SizeAsTarget, size);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override bool GetLeftIsChild(TLink node) =>
46             ↪ GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override void SetLeftIsChild(TLink node, bool value) =>
50             ↪ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override bool GetRightIsChild(TLink node) =>
54             ↪ GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override void SetRightIsChild(TLink node, bool value) =>
58             ↪ SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override sbyte GetBalance(TLink node) =>
62             ↪ GetBalanceValue(GetLinkReference(node).SizeAsTarget);
63     }
64 }

```

```

50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
    ↳ GetLinkReference(node).SizeAsTarget, value);
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
    ↳ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
    ↳ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
    ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
    ↳ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override void ClearNode(TLink node)
68 {
69     ref var link = ref GetLinkReference(node);
70     link.LeftAsTarget = Zero;
71     link.RightAsTarget = Zero;
72     link.SizeAsTarget = Zero;
73 }
74 }
75 }

```

1.37 ./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 unsafe override ref TLink GetLeftReference(TLink node) => ref
16             ↳ GetLinkReference(node).LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected unsafe 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)]

```

```

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

```

1.38 ./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 partial class ResizableDirectMemoryLinks<TLink> :
12         ↪ ResizableDirectMemoryLinksBase<TLink>
13     {
14         private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
15         private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
16         private byte* _header;
17         private byte* _links;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public ResizableDirectMemoryLinks(string address) : this(address, DefaultLinksSizeStep)
21         ↪ { }
22
23         /// <summary>
24         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
25         ↪ минимальным шагом расширения базы данных.
26         /// </summary>
27         /// <param name="address">Полный путь к файлу базы данных.</param>
28         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
29         ↪ байтах.</param>
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public ResizableDirectMemoryLinks(string address, long memoryReservationStep) : this(new
32         ↪ FileMappedResizableDirectMemory(address, memoryReservationStep),
33         ↪ memoryReservationStep) { }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
37         ↪ DefaultLinksSizeStep) { }
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
41         ↪ memoryReservationStep) : this(memory, memoryReservationStep,
42         ↪ Default<LinksConstants<TLink>>.Instance, true) { }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
46         ↪ memoryReservationStep, LinksConstants<TLink> constants, bool useAvlBasedIndex) :
47         ↪ base(memory, memoryReservationStep, constants)
48         {
49             if (useAvlBasedIndex)
50             {
51                 _createSourceTreeMethods = () => new
52                 ↪ LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
53                 _createTargetTreeMethods = () => new
54                 ↪ LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
55             }
56         }
57     }
58 }

```

```

43     else
44     {
45         _createSourceTreeMethods = () => new
46             ↳ LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
47         _createTargetTreeMethods = () => new
48             ↳ LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
49     }
50     Init(memory, memoryReservationStep);
51 }
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 protected override void SetPointers(IResizableDirectMemory memory)
54 {
55     _links = (byte*)memory.Pointer;
56     _header = _links;
57     SourcesTreeMethods = _createSourceTreeMethods();
58     TargetsTreeMethods = _createTargetTreeMethods();
59     UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
60 }
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override void ResetPointers()
63 {
64     base.ResetPointers();
65     _links = null;
66     _header = null;
67 }
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 protected override ref LinksHeader<TLink> GetHeaderReference() => ref
70     ↳ AsRef<LinksHeader<TLink>>(_header);
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
73     ↳ AsRef<RawLink<TLink>>(_links + (LinkSizeInBytes * ConvertToInt64(linkIndex)));
74 }
75 }

```

1.39 ./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 = LinksHeader<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 }

```



```

39     protected ILinksTreeMethods<TLink> TargetsTreeMethods;
40     protected ILinksTreeMethods<TLink> SourcesTreeMethods;
41     // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
42     ↪     нужно использовать не список а дерево, так как так можно быстрее проверить на
43     ↪     наличие связи внутри
44     protected ILinksListMethods<TLink> UnusedLinksListMethods;
45
46     /// <summary>
47     /// Возвращает общее число связей находящихся в хранилище.
48     /// </summary>
49     protected virtual TLink Total
50     {
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         get
53         {
54             ref var header = ref GetHeaderReference();
55             return Subtract(header.AllocatedLinks, header.FreeLinks);
56         }
57     }
58
59     public virtual LinksConstants<TLink> Constants
60     {
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         get;
63     }
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
67     ↪     memoryReservationStep, LinksConstants<TLink> constants)
68     {
69         _memory = memory;
70         _memoryReservationStep = memoryReservationStep;
71         Constants = constants;
72     }
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     protected ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
76     ↪     memoryReservationStep) : this(memory, memoryReservationStep,
77     ↪     Default<LinksConstants<TLink>>.Instance) { }
78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
81     {
82         if (memory.ReservedCapacity < memoryReservationStep)
83         {
84             memory.ReservedCapacity = memoryReservationStep;
85         }
86         SetPointers(_memory);
87         ref var header = ref GetHeaderReference();
88         // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
89         _memory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes) +
90         ↪     LinkHeaderSizeInBytes;
91         // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
92         header.ReservedLinks = ConvertToAddress((_memory.ReservedCapacity -
93         ↪     LinkHeaderSizeInBytes) / LinkSizeInBytes);
94     }
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     public virtual TLink Count(IList<TLink> restrictions)
98     {
99         // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
100         if (restrictions.Count == 0)
101         {
102             return Total;
103         }
104         var constants = Constants;
105         var any = constants.Any;
106         var index = restrictions[constants.IndexPart];
107         if (restrictions.Count == 1)
108         {
109             if (AreEqual(index, any))
110             {
111                 return Total;
112             }
113             return Exists(index) ? GetOne() : GetZero();
114         }
115         if (restrictions.Count == 2)
116         {
117             var value = restrictions[1];
118         }
119     }

```

```

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),
118             ↪ TargetsTreeMethods.CountUsages(value));
119     }
120     else
121     {
122         if (!Exists(index))
123         {
124             return GetZero();
125         }
126         if (AreEqual(value, any))
127         {
128             return GetOne();
129         }
130         ref var storedLinkValue = ref GetLinkReference(index);
131         if (AreEqual(storedLinkValue.Source, value) ||
132             ↪ AreEqual(storedLinkValue.Target, value))
133         {
134             return GetOne();
135         }
136         return GetZero();
137     }
138 }
139 if (restrictions.Count == 3)
140 {
141     var source = restrictions[constants.SourcePart];
142     var target = restrictions[constants.TargetPart];
143     if (AreEqual(index, any))
144     {
145         if (AreEqual(source, any) && AreEqual(target, any))
146         {
147             return Total;
148         }
149         else if (AreEqual(source, any))
150         {
151             return TargetsTreeMethods.CountUsages(target);
152         }
153         else if (AreEqual(target, any))
154         {
155             return SourcesTreeMethods.CountUsages(source);
156         }
157         else //if(source != Any && target != Any)
158         {
159             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
160             var link = SourcesTreeMethods.Search(source, target);
161             return AreEqual(link, constants.Null) ? GetZero() : GetOne();
162         }
163     }
164     else
165     {
166         if (!Exists(index))
167         {
168             return GetZero();
169         }
170         if (AreEqual(source, any) && AreEqual(target, any))
171         {
172             return GetOne();
173         }
174         ref var storedLinkValue = ref GetLinkReference(index);
175         if (!AreEqual(source, any) && !AreEqual(target, any))
176         {
177             if (AreEqual(storedLinkValue.Source, source) &&
178                 ↪ AreEqual(storedLinkValue.Target, target))
179             {
180                 return GetOne();
181             }
182             return GetZero();
183         }
184         var value = default(TLink);
185         if (AreEqual(source, any))
186         {
187             value = target;
188         }
189     }
190 }

```

```

186         if (AreEqual(target, any))
187         {
188             value = source;
189         }
190         if (AreEqual(storedLinkValue.Source, value) ||
191             ↪ AreEqual(storedLinkValue.Target, value))
192         {
193             return GetOne();
194         }
195         return GetZero();
196     }
197     throw new NotSupportedException("Другие размеры и способы ограничений не
198     ↪ поддерживаются.");
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         return @break;
216     }
217     var @continue = constants.Continue;
218     var any = constants.Any;
219     var index = restrictions[constants.IndexPart];
220     if (restrictions.Count == 1)
221     {
222         if (AreEqual(index, any))
223         {
224             return Each(handler, GetEmptyList());
225         }
226         if (!Exists(index))
227         {
228             return @continue;
229         }
230         return handler(GetLinkStruct(index));
231     }
232     if (restrictions.Count == 2)
233     {
234         var value = restrictions[1];
235         if (AreEqual(index, any))
236         {
237             if (AreEqual(value, any))
238             {
239                 return Each(handler, GetEmptyList());
240             }
241             if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
242             {
243                 return @break;
244             }
245             return Each(handler, new Link<TLink>(index, any, value));
246         }
247         else
248         {
249             if (!Exists(index))
250             {
251                 return @continue;
252             }
253             if (AreEqual(value, any))
254             {
255                 return handler(GetLinkStruct(index));
256             }
257             ref var storedLinkValue = ref GetLinkReference(index);
258             if (AreEqual(storedLinkValue.Source, value) ||
259                 AreEqual(storedLinkValue.Target, value))
260             {
261                 return handler(GetLinkStruct(index));
262             }

```

```

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, GetEmptyList());
274         }
275         else if (AreEqual(source, any))
276         {
277             return TargetsTreeMethods.EachUsage(target, handler);
278         }
279         else if (AreEqual(target, any))
280         {
281             return SourcesTreeMethods.EachUsage(source, handler);
282         }
283         else //if(source != Any && target != Any)
284         {
285             var link = SourcesTreeMethods.Search(source, target);
286             return AreEqual(link, constants.Null) ? @continue :
                ↪ handler(GetLinkStruct(link));
287         }
288     }
289     else
290     {
291         if (!Exists(index))
292         {
293             return @continue;
294         }
295         if (AreEqual(source, any) && AreEqual(target, any))
296         {
297             return handler(GetLinkStruct(index));
298         }
299         ref var storedLinkValue = ref GetLinkReference(index);
300         if (!AreEqual(source, any) && !AreEqual(target, any))
301         {
302             if (AreEqual(storedLinkValue.Source, source) &&
303                 AreEqual(storedLinkValue.Target, target))
304             {
305                 return handler(GetLinkStruct(index));
306             }
307             return @continue;
308         }
309         var value = default(TLink);
310         if (AreEqual(source, any))
311         {
312             value = target;
313         }
314         if (AreEqual(target, any))
315         {
316             value = source;
317         }
318         if (AreEqual(storedLinkValue.Source, value) ||
319             AreEqual(storedLinkValue.Target, value))
320         {
321             return handler(GetLinkStruct(index));
322         }
323         return @continue;
324     }
325 }
326 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↪ поддерживаются.");
327 }
328
329 /// <remarks>
330 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
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;

```

```

336     var @null = constants.Null;
337     var linkIndex = restrictions[constants.IndexPart];
338     ref var link = ref GetLinkReference(linkIndex);
339     ref var header = ref GetHeaderReference();
340     ref var firstAsSource = ref header.FirstAsSource;
341     ref var firstAsTarget = ref header.FirstAsTarget;
342     // Будет корректно работать только в том случае, если пространство выделенной связи
343     ↪ предварительно заполнено нулями
344     if (!AreEqual(link.Source, @null))
345     {
346         SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
347     }
348     if (!AreEqual(link.Target, @null))
349     {
350         TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
351     }
352     link.Source = substitution[constants.SourcePart];
353     link.Target = substitution[constants.TargetPart];
354     if (!AreEqual(link.Source, @null))
355     {
356         SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
357     }
358     if (!AreEqual(link.Target, @null))
359     {
360         TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
361     }
362     return linkIndex;
363 }
364
365 /// <remarks>
366 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
367 ↪ пространство
368 /// </remarks>
369 [MethodImpl(MethodImplOptions.AggressiveInlining)]
370 public virtual TLink Create(ICollection<TLink> restrictions)
371 {
372     ref var header = ref GetHeaderReference();
373     var freeLink = header.FirstFreeLink;
374     if (!AreEqual(freeLink, Constants.Null))
375     {
376         UnusedLinksListMethods.Detach(freeLink);
377     }
378     else
379     {
380         {
381             var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
382             if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
383             {
384                 throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
385             }
386             if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
387             {
388                 _memory.ReservedCapacity += _memory.ReservationStep;
389                 SetPointers(_memory);
390                 header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
391                 ↪ LinkSizeInBytes);
392             }
393             header.AllocatedLinks = Increment(header.AllocatedLinks);
394             _memory.UsedCapacity += LinkSizeInBytes;
395             freeLink = header.AllocatedLinks;
396         }
397     }
398     return freeLink;
399 }
400
401 [MethodImpl(MethodImplOptions.AggressiveInlining)]
402 public virtual void Delete(ICollection<TLink> restrictions)
403 {
404     ref var header = ref GetHeaderReference();
405     var link = restrictions[Constants.IndexPart];
406     if (LessThan(link, header.AllocatedLinks))
407     {
408         UnusedLinksListMethods.AttachAsFirst(link);
409     }
410     else if (AreEqual(link, header.AllocatedLinks))
411     {
412         header.AllocatedLinks = Decrement(header.AllocatedLinks);
413         _memory.UsedCapacity -= LinkSizeInBytes;
414         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
415         ↪ пока не дойдём до первой существующей связи
416         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)

```

```

411         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
412             ↪ IsUnusedLink(header.AllocatedLinks))
413         {
414             UnusedLinksListMethods.Detach(header.AllocatedLinks);
415             header.AllocatedLinks = Decrement(header.AllocatedLinks);
416             _memory.UsedCapacity -= LinkSizeInBytes;
417         }
418     }
419
420     [MethodImpl(MethodImplOptions.AggressiveInlining)]
421     public IList<TLink> GetLinkStruct(TLink linkIndex)
422     {
423         ref var link = ref GetLinkReference(linkIndex);
424         return new Link<TLink>(linkIndex, link.Source, link.Target);
425     }
426
427     /// <remarks>
428     /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
429     ↪ адрес реально поменялся
430     ///
431     /// Указатель this.links может быть в том же месте,
432     /// так как 0-я связь не используется и имеет такой же размер как Header,
433     /// поэтому header размещается в том же месте, что и 0-я связь
434     /// </remarks>
435     [MethodImpl(MethodImplOptions.AggressiveInlining)]
436     protected abstract void SetPointers(IResizableDirectMemory memory);
437
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)]

```

```

484     protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
485         ↪ _comparer.Compare(first, second) <= 0;
486
487     [MethodImpl(MethodImplOptions.AggressiveInlining)]
488     protected virtual bool GreaterThan(TLink first, TLink second) =>
489         ↪ _comparer.Compare(first, second) > 0;
490
491     [MethodImpl(MethodImplOptions.AggressiveInlining)]
492     protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
493         ↪ _comparer.Compare(first, second) >= 0;
494
495     [MethodImpl(MethodImplOptions.AggressiveInlining)]
496     protected virtual long ConvertToInt64(TLink value) =>
497         ↪ _addressToInt64Converter.Convert(value);
498
499     [MethodImpl(MethodImplOptions.AggressiveInlining)]
500     protected virtual TLink ConvertToAddress(long value) =>
501         ↪ _int64ToAddressConverter.Convert(value);
502
503     [MethodImpl(MethodImplOptions.AggressiveInlining)]
504     protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
505         ↪ second);
506
507     [MethodImpl(MethodImplOptions.AggressiveInlining)]
508     protected virtual TLink Subtract(TLink first, TLink second) =>
509         ↪ Arithmetic<TLink>.Subtract(first, second);
510
511     [MethodImpl(MethodImplOptions.AggressiveInlining)]
512     protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
513
514     [MethodImpl(MethodImplOptions.AggressiveInlining)]
515     protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
516
517     [MethodImpl(MethodImplOptions.AggressiveInlining)]
518     protected virtual IList<TLink> GetEmptyList() => Array.Empty<TLink>();
519
520     #region Disposable
521
522     protected override bool AllowMultipleDisposeCalls
523     {
524         [MethodImpl(MethodImplOptions.AggressiveInlining)]
525         get => true;
526     }
527
528     [MethodImpl(MethodImplOptions.AggressiveInlining)]
529     protected override void Dispose(bool manual, bool wasDisposed)
530     {
531         if (!wasDisposed)
532         {
533             ResetPointers();
534             _memory.DisposeIfPossible();
535         }
536     }
537
538     #endregion
539 }

```

1.40 ./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         {

```

```

20     _links = links;
21     _header = header;
22 }
23
24 [MethodImpl(MethodImplOptions.AggressiveInlining)]
25 protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
    ↳ AsRef<LinksHeader<TLink>>(_header);
26
27 [MethodImpl(MethodImplOptions.AggressiveInlining)]
28 protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
    ↳ AsRef<RawLink<TLink>>(_links + (RawLink<TLink>.SizeInBytes *
    ↳ _addressToInt64Converter.Convert(link)));
29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected override TLink GetSize() => GetHeaderReference().FreeLinks;
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
    ↳ element;
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
    ↳ element;
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override void SetPrevious(TLink element, TLink previous) =>
    ↳ GetLinkReference(element).Source = previous;
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected override void SetNext(TLink element, TLink next) =>
    ↳ GetLinkReference(element).Target = next;
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
59 }
60 }

```

1.41 ./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 ./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     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     TLink Search(TLink source, TLink target);
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     TLink EachUsage(TLink source, Func<IList<TLink>, TLink> handler);
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     void Detach(ref TLink firstAsSource, TLink linkIndex);
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     void Attach(ref TLink firstAsSource, TLink linkIndex);
24 }
25
26 }

```

1.43 ./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 =
13             ↳ EqualityComparer<TLink>.Default;
14
15         public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;
16
17         public TLink AllocatedLinks;
18         public TLink ReservedLinks;
19         public TLink FreeLinks;
20         public TLink FirstFreeLink;
21         public TLink FirstAsSource;
22         public TLink FirstAsTarget;
23         public TLink LastFreeLink;
24         public TLink Reserved8;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public override bool Equals(object obj) => obj is LinksHeader<TLink> linksHeader ?
28             ↳ Equals(linksHeader) : false;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool Equals(LinksHeader<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(FirstAsSource, other.FirstAsSource)
37             && _equalityComparer.Equals(FirstAsTarget, other.FirstAsTarget)
38             && _equalityComparer.Equals(LastFreeLink, other.LastFreeLink)
39             && _equalityComparer.Equals(Reserved8, other.Reserved8);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
43             ↳ FirstFreeLink, FirstAsSource, FirstAsTarget, LastFreeLink, Reserved8).GetHashCode();
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
47             ↳ left.Equals(right);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
51             ↳ !(left == right);
52     }
53 }

```

1.44 ./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 =
13         ↪ EqualityComparer<TLink>.Default;
14
15     public static readonly long SizeInBytes = Structure<RawLink<TLink>>.Size;
16
17     public TLink Source;
18     public TLink Target;
19     public TLink LeftAsSource;
20     public TLink RightAsSource;
21     public TLink SizeAsSource;
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 RawLink<TLink> link ? Equals(link) :
28         ↪ false;
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     public bool Equals(RawLink<TLink> other)
32     => _equalityComparer.Equals(Source, other.Source)
33     && _equalityComparer.Equals(Target, other.Target)
34     && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
35     && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
36     && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
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() => (Source, Target, LeftAsSource, RightAsSource,
43         ↪ SizeAsSource, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public static bool operator ==(RawLink<TLink> left, RawLink<TLink> right) =>
47         ↪ left.Equals(right);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     public static bool operator !=(RawLink<TLink> left, RawLink<TLink> right) => !(left ==
51         ↪ right);
52 }
53 }

```

1.45 ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs

```

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 :
10         ↪ LinksAvlBalancedTreeMethodsBase<ulong>
11     {
12         protected new readonly RawLink<ulong>* Links;
13         protected new readonly LinksHeader<ulong>* Header;
14
15         protected UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
16             ↪ RawLink<ulong>* links, LinksHeader<ulong>* header)
17             : base(constants, (byte*)links, (byte*)header)
18         {
19             Links = links;
20             Header = header;
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override ulong GetZero() => OUL;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override bool EqualToZero(ulong value) => value == OUL;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override bool AreEqual(ulong first, ulong second) => first == second;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override bool GreaterThanZero(ulong value) => value > OUL;
34
35     }
36 }

```

```

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
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override ulong Subtract(ulong first, ulong second) => first - second;
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
68 {
69     ref var firstLink = ref Links[first];
70     ref var secondLink = ref Links[second];
71     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↳ 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,
    ↳ secondLink.Source, secondLink.Target);
80 }
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected override ulong GetSizeValue(ulong value) => unchecked((value & 4294967264UL)
    ↳ >> 5);
84
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
    ↳ unchecked(storedValue & 31UL | (size & 134217727UL) << 5);
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 protected override bool GetLeftIsChildValue(ulong value) => unchecked((value & 16UL) >>
    ↳ 4 == 1UL);
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
    ↳ storedValue = unchecked(storedValue & 4294967279UL | (As<bool, byte>(ref value) &
    ↳ 1UL) << 4);
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected override bool GetRightIsChildValue(ulong value) => unchecked((value & 8UL) >>
    ↳ 3 == 1UL);
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

98     protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
99         ↪ storedValue = unchecked(storedValue & 4294967287UL | (As<bool, byte>(ref value) &
100         ↪ 1UL) << 3);
101
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
104     ↪ 0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
105     ↪ sbyte
106
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
109     ↪ storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
110     ↪ value & 3) & 7UL);
111
112     [MethodImpl(MethodImplOptions.AggressiveInlining)]
113     protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
114
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
117 }
118 }

```

1.46 ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSizeBalancedTreeMethodsBase.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 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             Links = links;
19             Header = header;
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetZero() => 0UL;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override bool EqualToZero(ulong value) => value == 0UL;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override bool AreEqual(ulong first, ulong second) => first == second;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override bool GreaterThanZero(ulong value) => value > 0UL;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override bool GreaterThan(ulong first, ulong second) => first > second;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
42         ↪ always true for ulong
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
46         ↪ always >= 0 for ulong
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
53         ↪ for ulong
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override bool LessThan(ulong first, ulong second) => first < second;

```

```

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 FirstIsToLeftOfSecond(ulong first, ulong second)
67 {
68     ref var firstLink = ref Links[first];
69     ref var secondLink = ref Links[second];
70     return FirstIsToLeftOfSecond(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 ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
88 }

```

1.47 ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
6 {
7     public unsafe class UInt64LinksSourcesAvlBalancedTreeMethods :
8         ↪ UInt64LinksAvlBalancedTreeMethodsBase
9     {
10         public UInt64LinksSourcesAvlBalancedTreeMethods(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) => GetSizeValue(Links[node].SizeAsSource);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

1.48 ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.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.Specific
6     {
7         public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
    ↳ UInt64LinksSizeBalancedTreeMethodsBase
8         {
9             public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
    ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
    ↳ { }
10
11             [MethodImpl(MethodImplOptions.AggressiveInlining)]
12             protected override ref ulong GetLeftReference(ulong node) => ref
    ↳ Links[node].LeftAsSource;

```

```

13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     protected override ref ulong GetRightReference(ulong node) => ref
15     ↪ Links[node].RightAsSource;
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
25     ↪ left;
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
29     ↪ right;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
36     ↪ size;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override ulong GetTreeRoot() => Header->FirstAsSource;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
46     ↪ ulong secondSource, ulong secondTarget)
47     => firstSource < secondSource || (firstSource == secondSource && firstTarget <
48     ↪ secondTarget);
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
52     ↪ ulong secondSource, ulong secondTarget)
53     => firstSource > secondSource || (firstSource == secondSource && firstTarget >
54     ↪ secondTarget);
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override void ClearNode(ulong node)
58     {
59         ref var link = ref Links[node];
60         link.LeftAsSource = OUL;
61         link.RightAsSource = OUL;
62         link.SizeAsSource = OUL;
63     }
64 }

```

1.49 ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.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) => Links[node].SizeAsTarget;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
41         ↪ size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
45         ↪ ulong secondSource, ulong secondTarget)
46         => firstSource < secondSource || (firstSource == secondSource && firstTarget <
47         ↪ secondTarget);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
51         ↪ ulong secondSource, ulong secondTarget)
52         => firstSource > secondSource || (firstSource == secondSource && firstTarget >
53         ↪ secondTarget);
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override void ClearNode(ulong node)
57         {
58             ref var link = ref Links[node];
59             link.LeftAsSource = OUL;
60             link.RightAsSource = OUL;
61             link.SizeAsSource = OUL;
62         }
63     }
64 }

```

```

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

```

1.50 ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.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.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,
  ↳ ulong secondSource, ulong secondTarget)
43     => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
  ↳ secondSource);

44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
  ↳ ulong secondSource, ulong secondTarget)
47     => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
  ↳ secondSource);

48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override void ClearNode(ulong node)
51     {
52         ref var link = ref Links[node];
53         link.LeftAsTarget = OUL;
54         link.RightAsTarget = OUL;
55         link.SizeAsTarget = OUL;
56     }
57 }
58 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Memory;
5 using Platform.Data.Doublets.ResizableDirectMemory.Generic;
6 using Platform.Singletons;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
11 {
12     public unsafe class UInt64ResizableDirectMemoryLinks : ResizableDirectMemoryLinksBase<ulong>
13     {

```

```

14 private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
15 private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
16 private LinksHeader<ulong>* _header;
17 private RawLink<ulong>* _links;
18
19 [MethodImpl(MethodImplOptions.AggressiveInlining)]
20 public UInt64ResizableDirectMemoryLinks(string address) : this(address,
    → DefaultLinksSizeStep) { }
21
22 /// <summary>
23 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
    → минимальным шагом расширения базы данных.
24 /// </summary>
25 /// <param name="address">Полный путь к файлу базы данных.</param>
26 /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
    → байтах.</param>
27 [MethodImpl(MethodImplOptions.AggressiveInlining)]
28 public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
    → this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
    → memoryReservationStep) { }
29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
    → DefaultLinksSizeStep) { }
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
    → memoryReservationStep) : this(memory, memoryReservationStep,
    → Default<LinksConstants<ulong>>.Instance, true) { }
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
    → memoryReservationStep, LinksConstants<ulong> constants, bool useAvlBasedIndex) :
    → base(memory, memoryReservationStep, constants)
38 {
39     if (useAvlBasedIndex)
40     {
41         _createSourceTreeMethods = () => new
            → UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
42         _createTargetTreeMethods = () => new
            → UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
43     }
44     else
45     {
46         _createSourceTreeMethods = () => new
            → UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
47         _createTargetTreeMethods = () => new
            → UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
48     }
49     Init(memory, memoryReservationStep);
50 }
51
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 protected override void SetPointers(IResizableDirectMemory memory)
54 {
55     _header = (LinksHeader<ulong>*)memory.Pointer;
56     _links = (RawLink<ulong>*)memory.Pointer;
57     SourcesTreeMethods = _createSourceTreeMethods();
58     TargetsTreeMethods = _createTargetTreeMethods();
59     UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
60 }
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override void ResetPointers()
64 {
65     base.ResetPointers();
66     _links = null;
67     _header = null;
68 }
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
    → _links[linkIndex];
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

77     protected override bool AreEqual(ulong first, ulong second) => first == second;
78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     protected override bool LessThan(ulong first, ulong second) => first < second;
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected override bool GreaterThan(ulong first, ulong second) => first > second;
87
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
90
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     protected override ulong GetZero() => 0UL;
93
94     [MethodImpl(MethodImplOptions.AggressiveInlining)]
95     protected override ulong GetOne() => 1UL;
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     protected override long ConvertToInt64(ulong value) => (long)value;
99
100    [MethodImpl(MethodImplOptions.AggressiveInlining)]
101    protected override ulong ConvertToAddress(long value) => (ulong)value;
102
103    [MethodImpl(MethodImplOptions.AggressiveInlining)]
104    protected override ulong Add(ulong first, ulong second) => first + second;
105
106    [MethodImpl(MethodImplOptions.AggressiveInlining)]
107    protected override ulong Subtract(ulong first, ulong second) => first - second;
108
109    [MethodImpl(MethodImplOptions.AggressiveInlining)]
110    protected override ulong Increment(ulong link) => ++link;
111
112    [MethodImpl(MethodImplOptions.AggressiveInlining)]
113    protected override ulong Decrement(ulong link) => --link;
114 }
115 }

```

1.52 ./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 ./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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 private void HalveSequence(ICollection<TLink> destination, ICollection<TLink> source, int length)
46 {
47     var loopedLength = length - (length % 2);
48     for (var i = 0; i < loopedLength; i += 2)
49     {
50         destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
51     }
52     if (length > loopedLength)
53     {
54         destination[length / 2] = source[length - 1];
55     }
56 }
57 }

```

1.54 ./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;

```

```

26 private static readonly TLink _one = Arithmetic.Increment(_zero);
27
28 private readonly IConverter<IList<TLink>, TLink> _baseConverter;
29 private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
30 private readonly TLink _minFrequencyToCompress;
31 private readonly bool _doInitialFrequenciesIncrement;
32 private Doublet<TLink> _maxDoublet;
33 private LinkFrequency<TLink> _maxDoubletData;
34
35 private struct HalfDoublet
36 {
37     public TLink Element;
38     public LinkFrequency<TLink> DoubletData;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
42     {
43         Element = element;
44         DoubletData = doubletData;
45     }
46
47     public override string ToString() => $"{Element}: ({DoubletData})";
48 }
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
    ↳ : this(links, baseConverter, doubletFrequenciesCache, _one, true)
52 {
53 }
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
    ↳ doInitialFrequenciesIncrement)
    ↳ : this(links, baseConverter, doubletFrequenciesCache, _one,
    ↳ doInitialFrequenciesIncrement)
57 {
58 }
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
    ↳ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
    ↳ : base(links)
62 {
63     _baseConverter = baseConverter;
64     _doubletFrequenciesCache = doubletFrequenciesCache;
65     if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
66     {
67         minFrequencyToCompress = _one;
68     }
69     _minFrequencyToCompress = minFrequencyToCompress;
70     _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
71     ResetMaxDoublet();
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 public override TLink Convert(IList<TLink> source) =>
    ↳ _baseConverter.Convert(Compress(source));
76
77 /// <remarks>
78 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
79 /// Faster version (doublets' frequencies dictionary is not recreated).
80 /// </remarks>
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 private IList<TLink> Compress(IList<TLink> sequence)
83 {
84     if (sequence.IsNullOrEmpty())
85     {
86         return null;
87     }
88     if (sequence.Count == 1)
89     {
90         return sequence;
91     }
92     if (sequence.Count == 2)
93     {
94         return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
95     }
96 }
97

```

```

98     }
99     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
100     var copy = new HalfDoublet[sequence.Count];
101     Doublet<TLink> doublet = default;
102     for (var i = 1; i < sequence.Count; i++)
103     {
104         doublet.Source = sequence[i - 1];
105         doublet.Target = sequence[i];
106         LinkFrequency<TLink> data;
107         if (_doInitialFrequenciesIncrement)
108         {
109             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
110         }
111         else
112         {
113             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
114             if (data == null)
115             {
116                 throw new NotSupportedException("If you ask not to increment
117                     ↪ frequencies, it is expected that all frequencies for the sequence
118                     ↪ are prepared.");
119             }
120             copy[i - 1].Element = sequence[i - 1];
121             copy[i - 1].DoubletData = data;
122             UpdateMaxDoublet(ref doublet, data);
123         }
124     }
125     copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
126     copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
127     if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
128     {
129         var newLength = ReplaceDoublets(copy);
130         sequence = new TLink[newLength];
131         for (int i = 0; i < newLength; i++)
132         {
133             sequence[i] = copy[i].Element;
134         }
135     }
136     return sequence;
137 }
138
139 /// <remarks>
140 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
141 /// </remarks>
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 private int ReplaceDoublets(HalfDoublet[] copy)
144 {
145     var oldLength = copy.Length;
146     var newLength = copy.Length;
147     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
148     {
149         var maxDoubletSource = _maxDoublet.Source;
150         var maxDoubletTarget = _maxDoublet.Target;
151         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
152         {
153             _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
154         }
155         var maxDoubletReplacementLink = _maxDoubletData.Link;
156         oldLength--;
157         var oldLengthMinusTwo = oldLength - 1;
158         // Substitute all usages
159         int w = 0, r = 0; // (r == read, w == write)
160         for (; r < oldLength; r++)
161         {
162             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
163                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
164             {
165                 if (r > 0)
166                 {
167                     var previous = copy[w - 1].Element;
168                     copy[w - 1].DoubletData.DecrementFrequency();
169                     copy[w - 1].DoubletData =
170                         ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
171                             ↪ maxDoubletReplacementLink);
172                 }
173                 if (r < oldLengthMinusTwo)
174                 {
175                     var next = copy[r + 2].Element;

```

```

171         copy[r + 1].DoubletData.DecrementFrequency();
172         copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
        ↪ xDoubletReplacementLink,
        ↪ next);
173     }
174     copy[w++].Element = maxDoubletReplacementLink;
175     r++;
176     newLength--;
177 }
178 else
179 {
180     copy[w++] = copy[r];
181 }
182 }
183 if (w < newLength)
184 {
185     copy[w] = copy[r];
186 }
187 oldLength = newLength;
188 ResetMaxDoublet();
189 UpdateMaxDoublet(copy, newLength);
190 }
191 return newLength;
192 }
193
194 [MethodImpl(MethodImplOptions.AggressiveInlining)]
195 private void ResetMaxDoublet()
196 {
197     _maxDoublet = new Doublet<TLink>();
198     _maxDoubletData = new LinkFrequency<TLink>();
199 }
200
201 [MethodImpl(MethodImplOptions.AggressiveInlining)]
202 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
203 {
204     Doublet<TLink> doublet = default;
205     for (var i = 1; i < length; i++)
206     {
207         doublet.Source = copy[i - 1].Element;
208         doublet.Target = copy[i].Element;
209         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
210     }
211 }
212
213 [MethodImpl(MethodImplOptions.AggressiveInlining)]
214 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
215 {
216     var frequency = data.Frequency;
217     var maxFrequency = _maxDoubletData.Frequency;
218     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
    ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
    ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
    ↪ _maxDoublet.Target)))
219     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
220         (_comparer.Compare(maxFrequency, frequency) < 0 ||
        ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
        ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
        ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
        ↪ better stability and better compression on sequent data and even on random
        ↪ numbers data (but gives collisions anyway) */
221     {
222         _maxDoublet = doublet;
223         _maxDoubletData = data;
224     }
225 }
226 }
227 }

```

1.55 ./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> : IConverter<IList<TLink>,
    ↪ TLink>

```

```

10 {
11     protected readonly ILinks<TLink> Links;
12
13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     protected LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     public abstract TLink Convert(ICollection<TLink> source);
18 }
19 }

```

1.56 ./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<ICollection<TLink>> _sequenceToItsLocalElementLevelsConverter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public OptimalVariantConverter(ILinks<TLink> links, IConverter<ICollection<TLink>>
20             ↪ sequenceToItsLocalElementLevelsConverter) : base(links)
21             => _sequenceToItsLocalElementLevelsConverter =
22                 ↪ sequenceToItsLocalElementLevelsConverter;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public override TLink Convert(ICollection<TLink> sequence)
26         {
27             var length = sequence.Count;
28             if (length == 1)
29             {
30                 return sequence[0];
31             }
32             var links = Links;
33             if (length == 2)
34             {
35                 return links.GetOrCreate(sequence[0], sequence[1]);
36             }
37             sequence = sequence.ToArray();
38             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
39             while (length > 2)
40             {
41                 var levelRepeat = 1;
42                 var currentLevel = levels[0];
43                 var previousLevel = levels[0];
44                 var skipOnce = false;
45                 var w = 0;
46                 for (var i = 1; i < length; i++)
47                 {
48                     if (_equalityComparer.Equals(currentLevel, levels[i]))
49                     {
50                         levelRepeat++;
51                         skipOnce = false;
52                         if (levelRepeat == 2)
53                         {
54                             sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
55                             var newLevel = i >= length - 1 ?
56                                 GetPreviousLowerThanCurrentOrCurrent(previousLevel,
57                                     ↪ currentLevel) :
58                                 i < 2 ?
59                                     GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
60                                     GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
61                                         ↪ currentLevel, levels[i + 1]);
62                             levels[w] = newLevel;
63                             previousLevel = currentLevel;
64                             w++;
65                             levelRepeat = 0;
66                             skipOnce = true;
67                         }
68                     }
69                     else if (i == length - 1)

```



```

64         {
65             sequence[w] = sequence[i];
66             levels[w] = levels[i];
67             w++;
68         }
69     }
70     else
71     {
72         currentLevel = levels[i];
73         levelRepeat = 1;
74         if (skipOnce)
75         {
76             skipOnce = false;
77         }
78         else
79         {
80             sequence[w] = sequence[i - 1];
81             levels[w] = levels[i - 1];
82             previousLevel = levels[w];
83             w++;
84         }
85         if (i == length - 1)
86         {
87             sequence[w] = sequence[i];
88             levels[w] = levels[i];
89             w++;
90         }
91     }
92     }
93     length = w;
94 }
95 return links.GetOrCreate(sequence[0], sequence[1]);
96 }
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
→ current, TLink next)
100 {
101     return _comparer.Compare(previous, next) > 0
102         ? _comparer.Compare(previous, current) < 0 ? previous : current
103         : _comparer.Compare(next, current) < 0 ? next : current;
104 }
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
→ _comparer.Compare(next, current) < 0 ? next : current;
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
→ => _comparer.Compare(previous, current) < 0 ? previous : current;
111 }
112 }

```

1.57 ./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<IList<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 IList<TLink> Convert(IList<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],
30     ↪ sequence[sequence.Count - 1]);
31     return levels;
32 }
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 public TLink GetFrequencyNumber(TLink source, TLink target) =>
35     ↪ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
36 }

```

1.58 ./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>,
9     ↪ ICriterionMatcher<TLink>
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
13
14        [MethodImpl(MethodImplOptions.AggressiveInlining)]
15        public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);
16    }

```

1.59 ./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    }

```

1.60 ./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Collections.Stacks;
4 using Platform.Data.Doublets.Sequences.HeightProviders;
5 using Platform.Data.Sequences;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Sequences
10 {

```

```

11 public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
    ↳ ISequenceAppender<TLink>
12 {
13     private static readonly EqualityComparer<TLink> _equalityComparer =
        ↳ EqualityComparer<TLink>.Default;
14
15     private readonly IStack<TLink> _stack;
16     private readonly ISequenceHeightProvider<TLink> _heightProvider;
17
18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
        ↳ ISequenceHeightProvider<TLink> heightProvider)
        : base(links)
20     {
21         _stack = stack;
22         _heightProvider = heightProvider;
23     }
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public TLink Append(TLink sequence, TLink appendant)
27     {
28         var cursor = sequence;
29         while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
30         {
31             var source = Links.GetSource(cursor);
32             var target = Links.GetTarget(cursor);
33             if (_equalityComparer.Equals(_heightProvider.Get(source),
        ↳ _heightProvider.Get(target)))
34             {
35                 break;
36             }
37             else
38             {
39                 _stack.Push(source);
40                 cursor = target;
41             }
42         }
43         var left = cursor;
44         var right = appendant;
45         while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
46         {
47             right = Links.GetOrCreate(left, right);
48             left = cursor;
49         }
50         return Links.GetOrCreate(left, right);
51     }
52 }
53 }
54 }

```

1.61 ./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>>>>
            ↳ _duplicateFragmentsProvider;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
            ↳ IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
            ↳ duplicateFragmentsProvider;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
19     }
20 }

```

1.62 ./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 ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
35             ↳ IList<TLink>>>
36         {
37             private readonly IListEqualityComparer<TLink> _listComparer;
38
39             public ItemEquilityComparer() => _listComparer =
40                 ↳ Default<IListEqualityComparer<TLink>>.Instance;
41
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
44                 ↳ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
45                 ↳ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
46                 ↳ right.Value);
47
48             [MethodImpl(MethodImplOptions.AggressiveInlining)]
49             public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
50                 ↳ (_listComparer.GetHashCode(pair.Key),
51                 ↳ _listComparer.GetHashCode(pair.Value)).GetHashCode();
52         }
53
54         private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
55         {
56             private readonly IListComparer<TLink> _listComparer;
57
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]
59             public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
60
61             [MethodImpl(MethodImplOptions.AggressiveInlining)]
62             public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
63                 ↳ KeyValuePair<IList<TLink>, IList<TLink>> right)
64             {
65                 var intermediateResult = _listComparer.Compare(left.Key, right.Key);
66                 if (intermediateResult == 0)
67                 {
68                     intermediateResult = _listComparer.Compare(left.Value, right.Value);
69                 }
70                 return intermediateResult;
71             }
72         }
73
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
76             : base(minimumStringSegmentLength: 2)
77         {
78             _links = links;
79             _sequences = sequences;
80         }
81
82         [MethodImpl(MethodImplOptions.AggressiveInlining)]
83         public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
84         {

```

```

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

```

```

143         {
144             var sequenceIndex =
145                 ↪ _uint64ToAddressConverter.Convert(partiallyMatchedSequence);
146             duplicates.Add(sequenceIndex);
147         }
148     }
149     duplicates.Sort();
150     return duplicates;
151 }
152 [MethodImpl(MethodImplOptions.AggressiveInlining)]
153 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
154 {
155     if (!(_links is ILinks<ulong> ulongLinks))
156     {
157         return;
158     }
159     var duplicatesKey = duplicatesItem.Key;
160     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
161     Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)}");
162     var duplicatesList = duplicatesItem.Value;
163     for (int i = 0; i < duplicatesList.Count; i++)
164     {
165         var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
166         var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
167             ↪ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
168             ↪ UnicodeMap.IsCharLink(link.Index) ?
169             ↪ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
170         Console.WriteLine(formattedSequenceStructure);
171         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
172             ↪ ulongLinks);
173         Console.WriteLine(sequenceString);
174     }
175     Console.WriteLine();
176 }
177 }
178 }

```

1.63 ./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);

```

```

38         return GetFrequency(ref doublet);
39     }
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
43     {
44         _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
45         return data;
46     }
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     public void IncrementFrequencies(IList<TLink> sequence)
50     {
51         for (var i = 1; i < sequence.Count; i++)
52         {
53             IncrementFrequency(sequence[i - 1], sequence[i]);
54         }
55     }
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
59     {
60         var doublet = new Doublet<TLink>(source, target);
61         return IncrementFrequency(ref doublet);
62     }
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     public void PrintFrequencies(IList<TLink> sequence)
66     {
67         for (var i = 1; i < sequence.Count; i++)
68         {
69             PrintFrequency(sequence[i - 1], sequence[i]);
70         }
71     }
72
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
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     public void ValidateFrequencies()
103     {
104         foreach (var entry in _doubletsCache)
105         {
106             var value = entry.Value;
107             var linkIndex = value.Link;
108             if (!_equalityComparer.Equals(linkIndex, default))
109             {
110                 var frequency = value.Frequency;
111                 var count = _frequencyCounter.Count(linkIndex);
112                 // TODO: Why `frequency` always greater than `count` by 1?
113                 if (((_comparer.Compare(frequency, count) > 0) &&
114                     ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
115                     || ((_comparer.Compare(count, frequency) > 0) &&
116                         ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))

```

```

114         {
115             throw new InvalidOperationException("Frequencies validation failed.");
116         }
117     }
118     //else
119     //{
120     //    if (value.Frequency > 0)
121     //    {
122     //        var frequency = value.Frequency;
123     //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
124     //        var count = _countLinkFrequency(linkIndex);
125     //
126     //        if ((frequency > count && frequency - count > 1) || (count > frequency
127     //            && count - frequency > 1))
128     //            throw new Exception("Frequencies validation failed.");
129     //    }
130     //}
131 }
132 }
133 }

```

1.64 ./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 ./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 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter

```

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 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs

```

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 }

```

```

44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     private bool VisitElement(TLink element)
46     {
47         if (_equalityComparer.Equals(element, _symbol))
48         {
49             _total = Arithmetic.Increment(_total);
50         }
51         return true;
52     }
53 }
54 }

```

1.68 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public TLink Count(TLink argument) => new
23             ↳ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
24             ↳ _markedSequenceMatcher, argument).Count();
25     }
26 }

```

1.69 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.

```

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 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.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 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     }

```

1.71 ./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             protected TLink _total;
20
21             [MethodImpl(MethodImplOptions.AggressiveInlining)]
22             public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
23             {
24                 _links = links;
25                 _symbol = symbol;
26                 _visits = new HashSet<TLink>();
27                 _total = default;
28             }
29
30             [MethodImpl(MethodImplOptions.AggressiveInlining)]
31             public TLink Count()
32             {
33                 if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
34                 {
35                     return _total;
36                 }
37                 CountCore(_symbol);
38                 return _total;
39             }
40
41             [MethodImpl(MethodImplOptions.AggressiveInlining)]
42             private void CountCore(TLink link)
43             {
44                 var any = _links.Constants.Any;
45                 if (_equalityComparer.Equals(_links.Count(any, link), default))
46                 {
47                     CountSequenceSymbolFrequency(link);
48                 }
49                 else
50                 {
51                     _links.Each(EachElementHandler, any, link);
52                 }
53             }
54
55             [MethodImpl(MethodImplOptions.AggressiveInlining)]
56             protected virtual void CountSequenceSymbolFrequency(TLink link)
57             {
58                 var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
59                     ↳ link, _symbol);
60                 _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
61             }
62
63             [MethodImpl(MethodImplOptions.AggressiveInlining)]
64             private TLink EachElementHandler(IList<TLink> doublet)
65             {
66                 var constants = _links.Constants;

```

```

65         var doubletIndex = doublet[constants.IndexPart];
66         if (_visits.Add(doubletIndex))
67         {
68             CountCore(doubletIndex);
69         }
70         return constants.Continue;
71     }
72 }
73 }

```

1.72 ./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> : LinksOperatorBase<TLink>,
11         ↪ ISequenceHeightProvider<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15
16         private readonly TLink _heightPropertyMarker;
17         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
18         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
19         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
20         private readonly IProperties<TLink, TLink, TLink> _propertyOperator;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public CachedSequenceHeightProvider(
24             ILinks<TLink> links,
25             ISequenceHeightProvider<TLink> baseHeightProvider,
26             IConverter<TLink> addressToUnaryNumberConverter,
27             IConverter<TLink> unaryNumberToAddressConverter,
28             TLink heightPropertyMarker,
29             IProperties<TLink, TLink, TLink> propertyOperator)
30             : base(links)
31         {
32             _heightPropertyMarker = heightPropertyMarker;
33             _baseHeightProvider = baseHeightProvider;
34             _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
35             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
36             _propertyOperator = propertyOperator;
37         }
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public TLink Get(TLink sequence)
41         {
42             TLink height;
43             var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
44             if (_equalityComparer.Equals(heightValue, default))
45             {
46                 height = _baseHeightProvider.Get(sequence);
47                 heightValue = _addressToUnaryNumberConverter.Convert(height);
48                 _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
49             }
50             else
51             {
52                 height = _unaryNumberToAddressConverter.Convert(heightValue);
53             }
54             return height;
55         }
56     }
57 }

```

1.73 ./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
13
14
15
16
17
18
19
20

```

```

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

```

1.74 ./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 ./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(ICollection<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         }
48     }
49 }

```

```

43     }
44     return indexed;
45 }
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 public bool MightContain(ICollection<TLink> sequence)
49 {
50     var indexed = true;
51     var i = sequence.Count;
52     while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
53     return indexed;
54 }
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 private bool IsIndexed(TLink source, TLink target)
58 {
59     var frequency = _cache.GetFrequency(source, target);
60     if (frequency == null)
61     {
62         return false;
63     }
64     return !_equalityComparer.Equals(frequency.Frequency, default);
65 }
66 }
67 }

```

1.76 ./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>,
11         ↳ ISequenceIndex<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↳ EqualityComparer<TLink>.Default;
15
16         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
17         private readonly IIncrementer<TLink> _frequencyIncrementer;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public FrequencyIncrementingSequenceIndex(ICollection<TLink> links, IProperty<TLink, TLink>
21             ↳ frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
22             : base(links)
23         {
24             _frequencyPropertyOperator = frequencyPropertyOperator;
25             _frequencyIncrementer = frequencyIncrementer;
26         }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public override bool Add(ICollection<TLink> sequence)
30         {
31             var indexed = true;
32             var i = sequence.Count;
33             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
34                 ↳ { }
35             for (; i >= 1; i--)
36             {
37                 Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
38             }
39             return indexed;
40         }
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         private bool IsIndexedWithIncrement(TLink source, TLink target)
44         {
45             var link = Links.SearchOrCreate(source, target);
46             var indexed = !_equalityComparer.Equals(link, default);
47             if (indexed)
48             {
49                 Increment(link);
50             }
51             return indexed;
52         }
53     }
54 }

```

```

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 ./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 ./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 ./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                     {
36                         links.GetOrCreate(sequence[i - 1], sequence[i]);
37                     }
38                 });
39             }
40             return indexed;
41         }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public bool MightContain(IList<TLink> sequence)
45         {
46             var links = _links.Unsync;
47             return _links.SyncRoot.ExecuteReadOperation(() =>
48             {
49                 var indexed = true;
50                 var i = sequence.Count;
51                 while (--i >= 1 && (indexed =
52                     ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
53                     ↳ sequence[i]), default))) { }
54                 return indexed;
55             });
56         }
57     }
58 }

```

1.80 ./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(IList<TLink> sequence) => false;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public virtual bool MightContain(IList<TLink> sequence) => true;
15     }
16 }

```


1.81 ./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                         }
81                     }
82                     last++;
83                 }
84                 return variants;
85             }
86         }
87     }
88 }

```

```

77         }
78         variants[last++] = variant;
79     }
80 }
81 }
82 return variants;
83 }
84
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 public List<ulong> CreateAllVariants1(params ulong[] sequence)
87 {
88     return _sync.ExecuteWriteOperation(() =>
89     {
90         if (sequence.IsNullOrEmpty())
91         {
92             return new List<ulong>();
93         }
94         Links.Unsync.EnsureLinkExists(sequence);
95         if (sequence.Length == 1)
96         {
97             return new List<ulong> { sequence[0] };
98         }
99         var results = new
100             ↳ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
101         return CreateAllVariants1Core(sequence, results);
102     });
103 }
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
107 {
108     if (sequence.Length == 2)
109     {
110         var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
111         if (link == Constants.Null)
112         {
113             throw new NotImplementedException("Creation cancellation is not
114                 ↳ implemented.");
115         }
116         results.Add(link);
117         return results;
118     }
119     var innerSequenceLength = sequence.Length - 1;
120     var innerSequence = new ulong[innerSequenceLength];
121     for (var li = 0; li < innerSequenceLength; li++)
122     {
123         var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
124         if (link == Constants.Null)
125         {
126             throw new NotImplementedException("Creation cancellation is not
127                 ↳ implemented.");
128         }
129         for (var isi = 0; isi < li; isi++)
130         {
131             innerSequence[isi] = sequence[isi];
132         }
133         innerSequence[li] = link;
134         for (var isi = li + 1; isi < innerSequenceLength; isi++)
135         {
136             innerSequence[isi] = sequence[isi + 1];
137         }
138         CreateAllVariants1Core(innerSequence, results);
139     }
140     return results;
141 }
142
143 #endregion
144
145 [MethodImpl(MethodImplOptions.AggressiveInlining)]
146 public HashSet<ulong> Each1(params ulong[] sequence)
147 {
148     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
149     Each1(link =>
150     {
151         if (!visitedLinks.Contains(link))
152         {
153             visitedLinks.Add(link); // изучить почему случаются повторы
154         }
155     });
156 }

```

```

152         return true;
153     }, sequence);
154     return visitedLinks;
155 }
156
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
159 {
160     if (sequence.Length == 2)
161     {
162         Links.Unsync.Each(sequence[0], sequence[1], handler);
163     }
164     else
165     {
166         var innerSequenceLength = sequence.Length - 1;
167         for (var li = 0; li < innerSequenceLength; li++)
168         {
169             var left = sequence[li];
170             var right = sequence[li + 1];
171             if (left == 0 && right == 0)
172             {
173                 continue;
174             }
175             var linkIndex = li;
176             ulong[] innerSequence = null;
177             Links.Unsync.Each(doublet =>
178             {
179                 if (innerSequence == null)
180                 {
181                     innerSequence = new ulong[innerSequenceLength];
182                     for (var isi = 0; isi < linkIndex; isi++)
183                     {
184                         innerSequence[isi] = sequence[isi];
185                     }
186                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
187                     {
188                         innerSequence[isi] = sequence[isi + 1];
189                     }
190                 }
191                 innerSequence[linkIndex] = doublet[Constants.IndexPart];
192                 Each1(handler, innerSequence);
193                 return Constants.Continue;
194             }, Constants.Any, left, right);
195         }
196     }
197 }
198
199 [MethodImpl(MethodImplOptions.AggressiveInlining)]
200 public HashSet<ulong> EachPart(params ulong[] sequence)
201 {
202     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
203     EachPartCore(link =>
204     {
205         var linkIndex = link[Constants.IndexPart];
206         if (!visitedLinks.Contains(linkIndex))
207         {
208             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
209         }
210         return Constants.Continue;
211     }, sequence);
212     return visitedLinks;
213 }
214
215 [MethodImpl(MethodImplOptions.AggressiveInlining)]
216 public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
217 {
218     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
219     EachPartCore(link =>
220     {
221         var linkIndex = link[Constants.IndexPart];
222         if (!visitedLinks.Contains(linkIndex))
223         {
224             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
225             return handler(new LinkAddress<LinkIndex>(linkIndex));
226         }
227         return Constants.Continue;
228     }, sequence);
229 }
230

```

```

231 [MethodImpl(MethodImplOptions.AggressiveInlining)]
232 private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
    ↳ sequence)
233 {
234     if (sequence.IsNullOrEmpty())
235     {
236         return;
237     }
238     Links.EnsureLinkIsAnyOrExists(sequence);
239     if (sequence.Length == 1)
240     {
241         var link = sequence[0];
242         if (link > 0)
243         {
244             handler(new LinkAddress<LinkIndex>(link));
245         }
246         else
247         {
248             Links.Each(Constants.Any, Constants.Any, handler);
249         }
250     }
251     else if (sequence.Length == 2)
252     {
253         // _links.Each(sequence[0], sequence[1], handler);
254         //   o_|      x_o ...
255         // x_|      |___|
256         Links.Each(sequence[1], Constants.Any, doublet =>
257         {
258             var match = Links.SearchOrDefault(sequence[0], doublet);
259             if (match != Constants.Null)
260             {
261                 handler(new LinkAddress<LinkIndex>(match));
262             }
263             return true;
264         });
265         // |_x      ... x_o
266         // |_o      |___|
267         Links.Each(Constants.Any, sequence[0], doublet =>
268         {
269             var match = Links.SearchOrDefault(doublet, sequence[1]);
270             if (match != 0)
271             {
272                 handler(new LinkAddress<LinkIndex>(match));
273             }
274             return true;
275         });
276         //           .x o_.
277         //           |___|
278         PartialStepRight(x => handler(x), sequence[0], sequence[1]);
279     }
280     else
281     {
282         throw new NotImplementedException();
283     }
284 }
285
286 [MethodImpl(MethodImplOptions.AggressiveInlining)]
287 private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
288 {
289     Links.Unsync.Each(Constants.Any, left, doublet =>
290     {
291         StepRight(handler, doublet, right);
292         if (left != doublet)
293         {
294             PartialStepRight(handler, doublet, right);
295         }
296         return true;
297     });
298 }
299
300 [MethodImpl(MethodImplOptions.AggressiveInlining)]
301 private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
302 {
303     Links.Unsync.Each(left, Constants.Any, rightStep =>
304     {
305         TryStepRightUp(handler, right, rightStep);
306         return true;
307     });

```

```

308 }
309
310 [MethodImpl(MethodImplOptions.AggressiveInlining)]
311 private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
    ↳ stepFrom)
312 {
313     var upStep = stepFrom;
314     var firstSource = Links.Unsync.GetTarget(upStep);
315     while (firstSource != right && firstSource != upStep)
316     {
317         upStep = firstSource;
318         firstSource = Links.Unsync.GetSource(upStep);
319     }
320     if (firstSource == right)
321     {
322         handler(new LinkAddress<LinkIndex>(stepFrom));
323     }
324 }
325
326 // TODO: Test
327 [MethodImpl(MethodImplOptions.AggressiveInlining)]
328 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
329 {
330     Links.Unsync.Each(right, Constants.Any, doublet =>
331     {
332         StepLeft(handler, left, doublet);
333         if (right != doublet)
334         {
335             PartialStepLeft(handler, left, doublet);
336         }
337         return true;
338     });
339 }
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
343 {
344     Links.Unsync.Each(Constants.Any, right, leftStep =>
345     {
346         TryStepLeftUp(handler, left, leftStep);
347         return true;
348     });
349 }
350
351 [MethodImpl(MethodImplOptions.AggressiveInlining)]
352 private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
353 {
354     var upStep = stepFrom;
355     var firstTarget = Links.Unsync.GetSource(upStep);
356     while (firstTarget != left && firstTarget != upStep)
357     {
358         upStep = firstTarget;
359         firstTarget = Links.Unsync.GetTarget(upStep);
360     }
361     if (firstTarget == left)
362     {
363         handler(new LinkAddress<LinkIndex>(stepFrom));
364     }
365 }
366
367 [MethodImpl(MethodImplOptions.AggressiveInlining)]
368 private bool StartsWith(ulong sequence, ulong link)
369 {
370     var upStep = sequence;
371     var firstSource = Links.Unsync.GetSource(upStep);
372     while (firstSource != link && firstSource != upStep)
373     {
374         upStep = firstSource;
375         firstSource = Links.Unsync.GetSource(upStep);
376     }
377     return firstSource == link;
378 }
379
380 [MethodImpl(MethodImplOptions.AggressiveInlining)]
381 private bool EndsWith(ulong sequence, ulong link)
382 {
383     var upStep = sequence;
384     var lastTarget = Links.Unsync.GetTarget(upStep);
385     while (lastTarget != link && lastTarget != upStep)

```

```

386     {
387         upStep = lastTarget;
388         lastTarget = Links.Unsync.GetTarget(upStep);
389     }
390     return lastTarget == link;
391 }
392
393 [MethodImpl(MethodImplOptions.AggressiveInlining)]
394 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
395 {
396     return _sync.ExecuteReadOperation(() =>
397     {
398         var results = new List<ulong>();
399         if (sequence.Length > 0)
400         {
401             Links.EnsureLinkExists(sequence);
402             var firstElement = sequence[0];
403             if (sequence.Length == 1)
404             {
405                 results.Add(firstElement);
406                 return results;
407             }
408             if (sequence.Length == 2)
409             {
410                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
411                 if (doublet != Constants.Null)
412                 {
413                     results.Add(doublet);
414                 }
415                 return results;
416             }
417             var linksInSequence = new HashSet<ulong>(sequence);
418             void handler(ICollection<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                     }
439                     return true;
440                 });
441                 if (filterPosition == sequence.Length)
442                 {
443                     results.Add(resultIndex);
444                 }
445             }
446             if (sequence.Length >= 2)
447             {
448                 StepRight(handler, sequence[0], sequence[1]);
449             }
450             var last = sequence.Length - 2;
451             for (var i = 1; i < last; i++)
452             {
453                 PartialStepRight(handler, sequence[i], sequence[i + 1]);
454             }
455             if (sequence.Length >= 3)
456             {
457                 StepLeft(handler, sequence[sequence.Length - 2],
458                     ↪ sequence[sequence.Length - 1]);
459             }
460         }
461         return results;
462     });
463 }

```

```

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             }
497             if (sequence.Length >= 3)
498             {
499                 StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
500                     ↪ sequence[sequence.Length - 1]);
501             }
502             return results;
503         });
504     }
505
506     public const int MaxSequenceFormatSize = 200;
507
508     [MethodImpl(MethodImplOptions.AggressiveInlining)]
509     public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
510     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
511
512     [MethodImpl(MethodImplOptions.AggressiveInlining)]
513     public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
514     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
515     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
516     ↪ elementToString, insertComma, knownElements));
517
518     [MethodImpl(MethodImplOptions.AggressiveInlining)]
519     private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
520     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
521     ↪ LinkIndex[] knownElements)
522     {
523         var linksInSequence = new HashSet<ulong>(knownElements);
524         //var entered = new HashSet<ulong>();
525         var sb = new StringBuilder();
526         sb.Append('{');
527         if (links.Exists(sequenceLink))
528         {
529             StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
530             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
531             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
532             {
533                 if (insertComma && sb.Length > 1)
534                 {
535                     sb.Append(',');
536                 }
537                 //if (entered.Contains(element))
538                 //{

```

```

531         // sb.Append('{');
532         // elementToString(sb, element);
533         // sb.Append('}');
534         //}
535         //else
536         elementToString(sb, element);
537         if (sb.Length < MaxSequenceFormatSize)
538         {
539             return true;
540         }
541         sb.Append(insertComma ? ", ..." : "...");
542         return false;
543     });
544 }
545 sb.Append('}');
546 return sb.ToString();
547 }
548
549 [MethodImpl(MethodImplOptions.AggressiveInlining)]
550 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
    ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
    ↪ knownElements);
551
552 [MethodImpl(MethodImplOptions.AggressiveInlining)]
553 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
    ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
    ↪ sequenceLink, elementToString, insertComma, knownElements));
554
555 [MethodImpl(MethodImplOptions.AggressiveInlining)]
556 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    ↪ LinkIndex[] knownElements)
557 {
558     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),
    ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
566         {
567             if (insertComma && sb.Length > 1)
568             {
569                 sb.Append(',');
570             }
571             if (entered.Contains(element))
572             {
573                 sb.Append('{');
574                 elementToString(sb, element);
575                 sb.Append('}');
576             }
577             else
578             {
579                 elementToString(sb, element);
580             }
581             if (sb.Length < MaxSequenceFormatSize)
582             {
583                 return true;
584             }
585             sb.Append(insertComma ? ", ..." : "...");
586             return false;
587         });
588     }
589     sb.Append('}');
590     return sb.ToString();
591 }
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,
        ↪ Links.Unsync.GetTarget,
612         x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
        ↪ x =>
        {
613             if (filterPosition == (sequence.Length - 1))
614             {
615                 return false;
616             }
617             if (filterPosition >= 0)
618             {
619                 if (x == sequence[filterPosition + 1])
620                 {
621                     filterPosition++;
622                 }
623                 else
624                 {
625                     return false;
626                 }
627             }
628             if (filterPosition < 0)
629             {
630                 if (x == sequence[0])
631                 {
632                     filterPosition = 0;
633                 }
634             }
635             return true;
636         }
637     });
638     if (filterPosition == (sequence.Length - 1))
639     {
640         filteredResults.Add(result);
641     }
642     return filteredResults;
643 }
644 return new List<ulong>();
645 });
646 }
647
648 [MethodImpl(MethodImplOptions.AggressiveInlining)]
649 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
650 {
651     return _sync.ExecuteReadOperation(() =>
652     {
653         if (sequence.Length > 0)
654         {
655             Links.EnsureLinkExists(sequence);
656             var results = new HashSet<ulong>();
657             for (var i = 0; i < sequence.Length; i++)
658             {
659                 AllUsagesCore(sequence[i], results);
660             }
661             var filteredResults = new HashSet<ulong>();
662             var matcher = new Matcher(this, sequence, filteredResults, null);
663             matcher.AddAllPartialMatchedToResults(results);
664             return filteredResults;
665         }
666         return new HashSet<ulong>();
667     });
668 }
669
670 [MethodImpl(MethodImplOptions.AggressiveInlining)]
671 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
672 ↪ params ulong[] sequence)
673 {
674     return _sync.ExecuteReadOperation(() =>
675     {

```

```

676         if (sequence.Length > 0)
677         {
678             Links.EnsureLinkExists(sequence);
679
680             var results = new HashSet<ulong>();
681             var filteredResults = new HashSet<ulong>();
682             var matcher = new Matcher(this, sequence, filteredResults, handler);
683             for (var i = 0; i < sequence.Length; i++)
684             {
685                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
686                 {
687                     return false;
688                 }
689             }
690             return true;
691         }
692         return true;
693     });
694 }
695
696 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
697 //{
698 //    return Sync.ExecuteReadOperation(() =>
699 //    {
700 //        if (sequence.Length > 0)
701 //        {
702 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
703
704 //            var firstResults = new HashSet<ulong>();
705 //            var lastResults = new HashSet<ulong>();
706
707 //            var first = sequence.First(x => x != LinksConstants.Any);
708 //            var last = sequence.Last(x => x != LinksConstants.Any);
709
710 //            AllUsagesCore(first, firstResults);
711 //            AllUsagesCore(last, lastResults);
712
713 //            firstResults.IntersectWith(lastResults);
714
715 //            //for (var i = 0; i < sequence.Length; i++)
716 //            //    AllUsagesCore(sequence[i], results);
717
718 //            var filteredResults = new HashSet<ulong>();
719 //            var matcher = new Matcher(this, sequence, filteredResults, null);
720 //            matcher.AddAllPartialMatchedToResults(firstResults);
721 //            return filteredResults;
722 //        }
723
724 //        return new HashSet<ulong>();
725 //    });
726 //}
727
728 [MethodImpl(MethodImplOptions.AggressiveInlining)]
729 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
730 {
731     return _sync.ExecuteReadOperation((Func<HashSet<ulong>>)(() =>
732     {
733         if (sequence.Length > 0)
734         {
735             ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
736                 ↪ (IList<ulong>)sequence);
737             var firstResults = new HashSet<ulong>();
738             var lastResults = new HashSet<ulong>();
739             var first = sequence.First(x => x != Constants.Any);
740             var last = sequence.Last(x => x != Constants.Any);
741             AllUsagesCore(first, firstResults);
742             AllUsagesCore(last, lastResults);
743             firstResults.IntersectWith(lastResults);
744             //for (var i = 0; i < sequence.Length; i++)
745             //    AllUsagesCore(sequence[i], results);
746             var filteredResults = new HashSet<ulong>();
747             var matcher = new Matcher(this, sequence, filteredResults, null);
748             matcher.AddAllPartialMatchedToResults(firstResults);
749             return filteredResults;
750         }
751         return new HashSet<ulong>();
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();
789             }
790             var filteredResults = new HashSet<ulong>();
791             var matcher = new Matcher(this, sequence, filteredResults, null,
792                 ↳ readAsElements);
793             matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
794                 ↳ x)); // OrderBy is a Hack
795             return filteredResults;
796         }
797         return new HashSet<ulong>();
798     });
799 }
800
801 // Does not work
802 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
803 //    ↳ params ulong[] sequence)
804 //{
805 //    var visited = new HashSet<ulong>();
806 //    var results = new HashSet<ulong>();
807 //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
808 //        ↳ true; }, readAsElements);
809 //    var last = sequence.Length - 1;
810 //    for (var i = 0; i < last; i++)
811 //    {
812 //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
813 //    }
814 //    return results;
815 //}
816
817 [MethodImpl(MethodImplOptions.AggressiveInlining)]
818 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
819 {
820     return _sync.ExecuteReadOperation(() =>
821     {
822         if (sequence.Length > 0)
823         {
824             Links.EnsureLinkExists(sequence);
825             //var firstElement = sequence[0];
826             //if (sequence.Length == 1)
827             //{
828                 //results.Add(firstElement);
829                 return results;
830             }
831         }
832     });
833 }

```

```

826 //if (sequence.Length == 2)
827 //{
828 //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
829 //    //if (doublet != Doublets.Links.Null)
830 //        results.Add(doublet);
831 //    return results;
832 //}
833 //var lastElement = sequence[sequence.Length - 1];
834 //Func<ulong, bool> handler = x =>
835 //{
836 //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
837 //        results.Add(x);
838 //    return true;
839 //};
840 //if (sequence.Length >= 2)
841 //    StepRight(handler, sequence[0], sequence[1]);
842 //var last = sequence.Length - 2;
843 //for (var i = 1; i < last; i++)
844 //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
845 //if (sequence.Length >= 3)
846 //    StepLeft(handler, sequence[sequence.Length - 2],
847 //        sequence[sequence.Length - 1]);
848 //if (sequence.Length == 1)
849 //    throw new NotImplementedException(); // all sequences, containing
850 //    this element?
851 //if (sequence.Length == 2)
852 //{
853 //    var results = new List<ulong>();
854 //    PartialStepRight(results.Add, sequence[0], sequence[1]);
855 //    return results;
856 //}
857 //var matches = new List<List<ulong>>();
858 //var last = sequence.Length - 1;
859 //for (var i = 0; i < last; i++)
860 //{
861 //    var results = new List<ulong>();
862 //    //StepRight(results.Add, sequence[i], sequence[i + 1]);
863 //    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
864 //    if (results.Count > 0)
865 //        matches.Add(results);
866 //    else
867 //        return results;
868 //    if (matches.Count == 2)
869 //    {
870 //        var merged = new List<ulong>();
871 //        for (var j = 0; j < matches[0].Count; j++)
872 //            for (var k = 0; k < matches[1].Count; k++)
873 //                CloseInnerConnections(merged.Add, matches[0][j],
874 //                    matches[1][k]);
875 //        if (merged.Count > 0)
876 //            matches = new List<List<ulong>> { merged };
877 //        else
878 //            return new List<ulong>();
879 //    }
880 //}
881 //if (matches.Count > 0)
882 //{
883 //    var usages = new HashSet<ulong>();
884 //    for (int i = 0; i < sequence.Length; i++)
885 //    {
886 //        AllUsagesCore(sequence[i], usages);
887 //    }
888 //    //for (int i = 0; i < matches[0].Count; i++)
889 //    //    AllUsagesCore(matches[0][i], usages);
890 //    //usages.UnionWith(matches[0]);
891 //    return usages.ToList();
892 //}
893 var firstLinkUsages = new HashSet<ulong>();
894 AllUsagesCore(sequence[0], firstLinkUsages);
895 firstLinkUsages.Add(sequence[0]);
896 //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
897 //    sequence[0] }; // or all sequences, containing this element?
898 //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
899 //    1).ToList();
900 var results = new HashSet<ulong>();

```

```

897         foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
898             ↪ firstLinkUsages, 1))
899         {
900             AllUsagesCore(match, results);
901         }
902         return results.ToList();
903     }
904     return new List<ulong>();
905 });
906 }
907
908 /// <remarks>
909 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
910 /// </remarks>
911 [MethodImpl(MethodImplOptions.AggressiveInlining)]
912 public HashSet<ulong> AllUsages(ulong link)
913 {
914     return _sync.ExecuteReadOperation(() =>
915     {
916         var usages = new HashSet<ulong>();
917         AllUsagesCore(link, usages);
918         return usages;
919     });
920 }
921
922 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
923 ↪ той связи с которой начинался поиск (STTTSSSTT),
924 // причём достаточно одного бита для хранения перехода влево или вправо
925 [MethodImpl(MethodImplOptions.AggressiveInlining)]
926 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
927 {
928     bool handler(ulong doublet)
929     {
930         if (usages.Add(doublet))
931         {
932             AllUsagesCore(doublet, usages);
933         }
934         return true;
935     }
936     Links.Unsync.Each(link, Constants.Any, handler);
937     Links.Unsync.Each(Constants.Any, link, handler);
938 }
939
940 [MethodImpl(MethodImplOptions.AggressiveInlining)]
941 public HashSet<ulong> AllBottomUsages(ulong link)
942 {
943     return _sync.ExecuteReadOperation(() =>
944     {
945         var visits = new HashSet<ulong>();
946         var usages = new HashSet<ulong>();
947         AllBottomUsagesCore(link, visits, usages);
948         return usages;
949     });
950 }
951
952 [MethodImpl(MethodImplOptions.AggressiveInlining)]
953 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
954 ↪ usages)
955 {
956     bool handler(ulong doublet)
957     {
958         if (visits.Add(doublet))
959         {
960             AllBottomUsagesCore(doublet, visits, usages);
961         }
962         return true;
963     }
964     if (Links.Unsync.Count(Constants.Any, link) == 0)
965     {
966         usages.Add(link);
967     }
968     else
969     {
970         Links.Unsync.Each(link, Constants.Any, handler);
971         Links.Unsync.Each(Constants.Any, link, handler);
972     }
973 }

```

```

972 [MethodImpl(MethodImplOptions.AggressiveInlining)]
973 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
974 {
975     if (Options.UseSequenceMarker)
976     {
977         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
978             ↪ Options.MarkedSequenceMatcher, symbol);
979         return counter.Count();
980     }
981     else
982     {
983         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
984             ↪ symbol);
985         return counter.Count();
986     }
987 }
988 [MethodImpl(MethodImplOptions.AggressiveInlining)]
989 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
990     ↪ LinkIndex> outerHandler)
991 {
992     bool handler(ulong doublet)
993     {
994         if (usages.Add(doublet))
995         {
996             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
997             {
998                 return false;
999             }
1000             if (!AllUsagesCore1(doublet, usages, outerHandler))
1001             {
1002                 return false;
1003             }
1004         }
1005         return true;
1006     }
1007     return Links.Unsync.Each(link, Constants.Any, handler)
1008         && Links.Unsync.Each(Constants.Any, link, handler);
1009 }
1010 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1011 public void CalculateAllUsages(ulong[] totals)
1012 {
1013     var calculator = new AllUsagesCalculator(Links, totals);
1014     calculator.Calculate();
1015 }
1016 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1017 public void CalculateAllUsages2(ulong[] totals)
1018 {
1019     var calculator = new AllUsagesCalculator2(Links, totals);
1020     calculator.Calculate();
1021 }
1022 private class AllUsagesCalculator
1023 {
1024     private readonly SynchronizedLinks<ulong> _links;
1025     private readonly ulong[] _totals;
1026
1027     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1028     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1029     {
1030         _links = links;
1031         _totals = totals;
1032     }
1033
1034     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1035     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1036         ↪ CalculateCore);
1037
1038     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1039     private bool CalculateCore(ulong link)
1040     {
1041         if (_totals[link] == 0)
1042         {
1043             var total = 1UL;
1044             _totals[link] = total;
1045             var visitedChildren = new HashSet<ulong>();
1046             bool linkCalculator(ulong child)

```

```

1047         {
1048             if (link != child && visitedChildren.Add(child))
1049             {
1050                 total += _totals[child] == 0 ? 1 : _totals[child];
1051             }
1052             return true;
1053         }
1054         _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1055         _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1056         _totals[link] = total;
1057     }
1058     return true;
1059 }
1060
1061 private class AllUsagesCalculator2
1062 {
1063     private readonly SynchronizedLinks<ulong> _links;
1064     private readonly ulong[] _totals;
1065
1066     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1067     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1068     {
1069         _links = links;
1070         _totals = totals;
1071     }
1072
1073     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1074     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1075         ↪ CalculateCore);
1076
1077     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1078     private bool IsElement(ulong link)
1079     {
1080         // _linksInSequence.Contains(link) ||
1081         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1082             ↪ link;
1083     }
1084
1085     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1086     private bool CalculateCore(ulong link)
1087     {
1088         // TODO: Проработать защиту от заикливания
1089         // Основано на SequenceWalker.WalkLeft
1090         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1091         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1092         Func<ulong, bool> isElement = IsElement;
1093         void visitLeaf(ulong parent)
1094         {
1095             if (link != parent)
1096             {
1097                 _totals[parent]++;
1098             }
1099         }
1100         void visitNode(ulong parent)
1101         {
1102             if (link != parent)
1103             {
1104                 _totals[parent]++;
1105             }
1106         }
1107         var stack = new Stack();
1108         var element = link;
1109         if (isElement(element))
1110         {
1111             visitLeaf(element);
1112         }
1113         else
1114         {
1115             while (true)
1116             {
1117                 if (isElement(element))
1118                 {
1119                     if (stack.Count == 0)
1120                     {
1121                         break;
1122                     }
1123                     element = stack.Pop();
1124                     var source = getSource(element);

```

```

1124         var target = getTarget(element);
1125         // 06pa6oрка элемеHта
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 }
1144 _totals[link]++;
1145 return true;
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(ICollection<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 зацикливания
1237     }
1238
1239     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1240     public bool Collect(ulong link)
1241     {
1242         if (_enter.Add(link))
1243         {
1244             if (_intersectWith.Contains(link))
1245             {
1246                 _usages.Add(link);
1247             }
1248             _links.Unsync.Each(link, _links.Constants.Any, Collect);
1249             _links.Unsync.Each(_links.Constants.Any, link, Collect);
1250         }
1251         return true;
1252     }
1253 }
1254
1255 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1256 private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
↪ 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
↪ right)
1264 {
1265     // Direct
1266     if (left == right)
1267     {
1268         handler(new LinkAddress<LinkIndex>(left));
1269     }
1270     var doublet = Links.Unsync.SearchOrDefault(left, right);
1271     if (doublet != Constants.Null)
1272     {
1273         handler(new LinkAddress<LinkIndex>(doublet));
1274     }
1275     // Inner
1276     CloseInnerConnections(handler, left, right);
1277     // Outer
1278     StepLeft(handler, left, right);
1279     StepRight(handler, left, right);

```



```

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
1366 // ↪ несколько раз в последовательности)
1367 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1368 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1369 {
1370     return _sync.ExecuteReadOperation(() =>
1371     {
1372         var results = new HashSet<ulong>();
1373         if (linksToConnect.Length > 0)
1374         {
1375             Links.EnsureLinkExists(linksToConnect);
1376             AllUsagesCore(linksToConnect[0], results);
1377             for (var i = 1; i < linksToConnect.Length; i++)
1378             {
1379                 var next = new HashSet<ulong>();
1380                 AllUsagesCore(linksToConnect[i], next);
1381                 results.IntersectWith(next);
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     }
1433     return results;
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         return results.GetSetUInt64Indices();
1459     });
1460 }
1461
1462 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1463 private static ulong[] Simplify(ulong[] sequence)
1464 {
1465     // Считаем новый размер последовательности
1466     long newLength = 0;
1467     var zeroOrManyStepped = false;
1468     for (var i = 0; i < sequence.Length; i++)
1469     {
1470         if (sequence[i] == ZeroOrMany)
1471         {
1472             if (zeroOrManyStepped)
1473             {
1474                 continue;
1475             }
1476             zeroOrManyStepped = true;
1477         }
1478         else
1479         {
1480             //if (zeroOrManyStepped) Is it efficient?
1481             zeroOrManyStepped = false;
1482         }
1483         newLength++;
1484     }
1485     // Строим новую последовательность
1486     zeroOrManyStepped = false;
1487     var newSequence = new ulong[newLength];
1488     long j = 0;
1489     for (var i = 0; i < sequence.Length; i++)
1490     {
1491         //var current = zeroOrManyStepped;
1492         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1493         //if (current && zeroOrManyStepped)
1494         //    continue;
1495         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1496         //if (zeroOrManyStepped && newZeroOrManyStepped)
1497         //    continue;
1498         //zeroOrManyStepped = newZeroOrManyStepped;
1499         if (sequence[i] == ZeroOrMany)
1500         {

```

```

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
1534
1535 //public static void DeleteSequence(Link sequence)
1536 //{
1537 //}
1538
1539 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1540 public List<ulong> CollectMatchingSequences(ulong[] links)
1541 {
1542     if (links.Length == 1)
1543     {
1544         throw new Exception("Подпоследовательности с одним элементом не
1545             ↪ поддерживаются.");
1546     }
1547     var leftBound = 0;
1548     var rightBound = links.Length - 1;
1549     var left = links[leftBound++];
1550     var right = links[rightBound--];
1551     var results = new List<ulong>();
1552     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1553     return results;
1554 }
1555
1556 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1557 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1558     ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1559 {
1560     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1561     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1562     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1563     {
1564         var nextLeftLink = middleLinks[leftBound];
1565         var elements = GetRightElements(leftLink, nextLeftLink);
1566         if (leftBound <= rightBound)
1567         {
1568             for (var i = elements.Length - 1; i >= 0; i--)
1569             {
1570                 var element = elements[i];
1571                 if (element != 0)
1572                 {
1573                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1574                         ↪ rightLink, rightBound, ref results);
1575                 }
1576             }
1577         }
1578     }
1579 }

```

```

1574     else
1575     {
1576         for (var i = elements.Length - 1; i >= 0; i--)
1577         {
1578             var element = elements[i];
1579             if (element != 0)
1580             {
1581                 results.Add(element);
1582             }
1583         }
1584     }
1585 }
1586 else
1587 {
1588     var nextRightLink = middleLinks[rightBound];
1589     var elements = GetLeftElements(rightLink, nextRightLink);
1590     if (leftBound <= rightBound)
1591     {
1592         for (var i = elements.Length - 1; i >= 0; i--)
1593         {
1594             var element = elements[i];
1595             if (element != 0)
1596             {
1597                 CollectMatchingSequences(leftLink, leftBound, middleLinks,
1598                                         ↪ elements[i], rightBound - 1, ref results);
1599             }
1600         }
1601     }
1602     else
1603     {
1604         for (var i = elements.Length - 1; i >= 0; i--)
1605         {
1606             var element = elements[i];
1607             if (element != 0)
1608             {
1609                 results.Add(element);
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     return true;
1665 });
1666 return added > 0;
1667 }
1668
1669 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1670 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1671 {
1672     var result = new ulong[5];
1673     TryStepLeft(startLink, leftLink, result, 0);
1674     Links.Each(startLink, Constants.Any, couple =>
1675     {
1676         if (couple != startLink)
1677         {
1678             if (TryStepLeft(couple, leftLink, result, 2))
1679             {
1680                 return false;
1681             }
1682         }
1683         return true;
1684     });
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             return true;
1719         });
1720     return added > 0;
1721 }
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));
1765         _results = results;
1766         _pattern = CreateDetailedPattern();
1767     }
1768
1769     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1770     protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1771         ↳ base.IsElement(link);
1772
1773     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1774     public bool PatternMatch(LinkIndex sequenceToMatch)
1775     {
1776         _patternPosition = 0;
1777         _sequencePosition = 0;
1778         foreach (var part in Walk(sequenceToMatch))
1779         {
1780             if (!PatternMatchCore(part))
1781             {
1782                 break;
1783             }
1784         }
1785         return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1786             ↳ - 1 && _pattern[_patternPosition].Start == 0);
1787     }
1788
1789     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1790     private List<PatternBlock> CreateDetailedPattern()
1791     {
1792         var pattern = new List<PatternBlock>();
1793         var patternBlock = new PatternBlock();
1794         for (var i = 0; i < _patternSequence.Length; i++)
1795         {
1796             if (patternBlock.Type == PatternBlockType.Undefined)
1797             {
1798                 if (_patternSequence[i] == _sequences.Constants.Any)
1799                 {
1800                     patternBlock.Type = PatternBlockType.Gap;
1801                     patternBlock.Start = 1;
1802                     patternBlock.Stop = 1;
1803                 }
1804                 else if (_patternSequence[i] == ZeroOrMany)
1805                 {
1806                     patternBlock.Type = PatternBlockType.Gap;
1807                     patternBlock.Start = 0;
1808                 }
1809             }
1810             else
1811             {
1812                 patternBlock.Type = PatternBlockType.Elements;
1813                 patternBlock.Start = i;
1814                 patternBlock.Stop = i + 1;
1815             }
1816             pattern.Add(patternBlock);
1817             patternBlock = new PatternBlock();
1818         }
1819     }

```



```

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 (regex[0] == '\0')
1886 //         return 1;
1887 //     if (regex[1] == '*')
1888 //         return matchstar(regex[0], regex + 2, text);
1889 //     if (regex[0] == '$' && regex[1] == '\0')
1890 //         return *text == '\0';
1891 //     if (*text != '\0' && (regex[0] == '.' || regex[0] == *text))
1892 //         return matchhere(regex + 1, text + 1);
1893 //     return 0;
1894 // }
1895
1896 // matchstar: search for c*regex at beginning of text
1897 // int matchstar(int c, char* regex, char* text)
1898 // {
1899 //     do
1900 //     {
1901 //         /* a * matches zero or more instances */
1902 //         if (matchhere(regex, 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
1921 //             break;
1922 //     }
1923 //     if (maximumGap < mininumGap)
1924 //         maximumGap = mininumGap;
1925 // }
1926
1927 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1928 private bool PatternMatchCore(LinkIndex element)
1929 {
1930     if (_patternPosition >= _pattern.Count)
1931     {
1932         _patternPosition = -2;
1933         return false;
1934     }
1935     var currentPatternBlock = _pattern[_patternPosition];
1936     if (currentPatternBlock.Type == PatternBlockType.Gap)
1937     {
1938         // var currentMatchingBlockLength = (_sequencePosition -
1939         //     ↪ _lastMatchedBlockPosition);
1940         if (_sequencePosition < currentPatternBlock.Start)
1941         {
1942             _sequencePosition++;
1943             return true; // Двигаемся дальше
1944         }
1945         // Это последний блок
1946         if (_pattern.Count == _patternPosition + 1)
1947         {
1948             _patternPosition++;
1949             _sequencePosition = 0;
1950             return false; // Полное соответствие
1951         }
1952         else
1953         {
1954             if (_sequencePosition > currentPatternBlock.Stop)
1955             {
1956                 return false; // Соответствие невозможно
1957             }
1958             var nextPatternBlock = _pattern[_patternPosition + 1];
1959             if (_patternSequence[nextPatternBlock.Start] == element)
1960             {
1961                 if (nextPatternBlock.Start < nextPatternBlock.Stop)

```

```

1960         {
1961             _patternPosition++;
1962             _sequencePosition = 1;
1963         }
1964         else
1965         {
1966             _patternPosition += 2;
1967             _sequencePosition = 0;
1968         }
1969     }
1970 }
1971 }
1972 else // currentPatternBlock.Type == PatternBlockType.Elements
1973 {
1974     var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1975     if (_patternSequence[patternElementPosition] != element)
1976     {
1977         return false; // Соответствие невозможно
1978     }
1979     if (patternElementPosition == currentPatternBlock.Stop)
1980     {
1981         _patternPosition++;
1982         _sequencePosition = 0;
1983     }
1984     else
1985     {
1986         _sequencePosition++;
1987     }
1988 }
1989 return true;
1990 //if (_patternSequence[_patternPosition] != element)
1991 //    return false;
1992 //else
1993 //{
1994 //    _sequencePosition++;
1995 //    _patternPosition++;
1996 //    return true;
1997 //}
1998 //if (_filterPosition == _patternSequence.Length)
1999 //{
2000 //    _filterPosition = -2; // Длиннее чем нужно
2001 //    return false;
2002 //}
2003 //if (element != _patternSequence[_filterPosition])
2004 //{
2005 //    _filterPosition = -1;
2006 //    return false; // Начинается иначе
2007 //}
2008 //_filterPosition++;
2009 //if (_filterPosition == (_patternSequence.Length - 1))
2010 //    return false;
2011 //if (_filterPosition >= 0)
2012 //{
2013 //    if (element == _patternSequence[_filterPosition + 1])
2014 //        _filterPosition++;
2015 //    else
2016 //        return false;
2017 //}
2018 //if (_filterPosition < 0)
2019 //{
2020 //    if (element == _patternSequence[0])
2021 //        _filterPosition = 0;
2022 //}
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 ./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.
34     ///
35     /// Рост последовательности слева и справа.
36     /// Поиск со звёздочкой.
37     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
38     /// так же проблема может быть решена при реализации дистанционных триггеров.
39     /// Нужны ли уникальные указатели вообще?
40     /// Что если обращение к информации будет происходить через содержимое всегда?
41     ///
42     /// Писать тесты.
43     ///
44     ///
45     /// Можно убрать зависимость от конкретной реализации Links,
46     /// на зависимость от абстрактного элемента, который может быть представлен несколькими
47     /// ↪ способами.
48     ///
49     /// Можно ли как-то сделать один общий интерфейс
50     ///
51     ///
52     /// Блокчейн и/или гит для распределённой записи транзакций.
53     ///
54     /// </remarks>
55     public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
56     ↪ (после завершения реализации Sequences)
57     {
58         /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
59         ↪ связей.</summary>
60         public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
61
62         public SequencesOptions<LinkIndex> Options { get; }
63         public SynchronizedLinks<LinkIndex> Links { get; }
64         private readonly ISynchronization _sync;
65
66         public LinksConstants<LinkIndex> Constants { get; }
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
70         {
71             Links = links;
72             _sync = links.SyncRoot;
73             Options = options;
74             Options.ValidateOptions();
75         }
76     }
77 }
```

```

68     Options.InitOptions(Links);
69     Constants = links.Constants;
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
    ↳ SequencesOptions<LinkIndex>()) { }
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 public bool IsSequence(LinkIndex sequence)
77 {
78     return _sync.ExecuteReadOperation(() =>
79     {
80         if (Options.UseSequenceMarker)
81         {
82             return Options.MarkedSequenceMatcher.IsMatched(sequence);
83         }
84         return !Links.Unsync.IsPartialPoint(sequence);
85     });
86 }
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 private LinkIndex GetSequenceByElements(LinkIndex sequence)
90 {
91     if (Options.UseSequenceMarker)
92     {
93         return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
94     }
95     return sequence;
96 }
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 private LinkIndex GetSequenceElements(LinkIndex sequence)
100 {
101     if (Options.UseSequenceMarker)
102     {
103         var linkContents = new Link<ulong>(Links.GetLink(sequence));
104         if (linkContents.Source == Options.SequenceMarkerLink)
105         {
106             return linkContents.Target;
107         }
108         if (linkContents.Target == Options.SequenceMarkerLink)
109         {
110             return linkContents.Source;
111         }
112     }
113     return sequence;
114 }
115
116 #region Count
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public LinkIndex Count(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 #endregion
175 #region Create
176 [MethodImpl(MethodImplOptions.AggressiveInlining)]
177 public LinkIndex Create(ICollection<LinkIndex> restrictions)
178 {
179     return _sync.ExecuteWriteOperation(() =>
180     {
181         if (restrictions.IsNullOrEmpty())
182         {
183             return Constants.Null;
184         }
185         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
186         return CreateCore(restrictions);
187     });
188 }
189 [MethodImpl(MethodImplOptions.AggressiveInlining)]
190 private LinkIndex CreateCore(ICollection<LinkIndex> restrictions)
191 {
192     LinkIndex[] sequence = restrictions.SkipFirst();
193     if (Options.UseIndex)
194     {
195         Options.Index.Add(sequence);
196     }
197     var sequenceRoot = default(LinkIndex);
198     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
199     {
200         var matches = Each(restrictions);
201         if (matches.Count > 0)
202         {
203             sequenceRoot = matches[0];
204         }
205     }
206     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
207     {
208         return CompactCore(sequence);
209     }
210     if (sequenceRoot == default)
211     {
212         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
213     }
214     if (Options.UseSequenceMarker)
215     {
216         return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
217     }
218     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
219 }
220
221
222

```

```

223 #endregion
224
225 #region Each
226
227 [MethodImpl(MethodImplOptions.AggressiveInlining)]
228 public List<LinkIndex> Each(IList<LinkIndex> sequence)
229 {
230     var results = new List<LinkIndex>();
231     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
232     Each(filler.AddFirstAndReturnConstant, sequence);
233     return results;
234 }
235
236 [MethodImpl(MethodImplOptions.AggressiveInlining)]
237 public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
238     ↪ 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
259                 {
260                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
261                         ↪ any));
262                 }
263             }
264             if (Options.UseSequenceMarker)
265             {
266                 var sequenceLinkValues = Links.Unsync.GetLink(link);
267                 if (sequenceLinkValues[Constants.SourcePart] ==
268                     ↪ Options.SequenceMarkerLink)
269                 {
270                     link = sequenceLinkValues[Constants.TargetPart];
271                 }
272             }
273             var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
274             sequence[0] = link;
275             return handler(sequence);
276         }
277         else if (restrictions.Count == 2)
278         {
279             throw new NotImplementedException();
280         }
281         else if (restrictions.Count == 3)
282         {
283             return Links.Unsync.Each(handler, restrictions);
284         }
285         else
286         {
287             var sequence = restrictions.SkipFirst();
288             if (Options.UseIndex && !Options.Index.MightContain(sequence))
289             {
290                 return Constants.Break;
291             }
292             return EachCore(handler, sequence);
293         }
294     });
295 }
296
297 [MethodImpl(MethodImplOptions.AggressiveInlining)]
298 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
299     ↪ values)

```

```

296 {
297     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
298     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
299     ↪ Id.
300     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
301     ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
302     ↪ matcher.HandleFullMatched;
303     //if (sequence.Length >= 2)
304     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
305     {
306         return Constants.Break;
307     }
308     var last = values.Count - 2;
309     for (var i = 1; i < last; i++)
310     {
311         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
312         ↪ Constants.Continue)
313         {
314             return Constants.Break;
315         }
316     }
317     if (values.Count >= 3)
318     {
319         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
320         ↪ != Constants.Continue)
321         {
322             return Constants.Break;
323         }
324     }
325     return Constants.Continue;
326 }
327
328 [MethodImpl(MethodImplOptions.AggressiveInlining)]
329 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
330 ↪ left, LinkIndex right)
331 {
332     return Links.Unsync.Each(doublet =>
333     {
334         var doubletIndex = doublet[Constants.IndexPart];
335         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
336         {
337             return Constants.Break;
338         }
339         if (left != doubletIndex)
340         {
341             return PartialStepRight(handler, doubletIndex, right);
342         }
343         return Constants.Continue;
344     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
345 }
346
347 [MethodImpl(MethodImplOptions.AggressiveInlining)]
348 private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
349 ↪ LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
350 ↪ rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
351 ↪ Constants.Any));
352
353 [MethodImpl(MethodImplOptions.AggressiveInlining)]
354 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
355 ↪ right, LinkIndex stepFrom)
356 {
357     var upStep = stepFrom;
358     var firstSource = Links.Unsync.GetTarget(upStep);
359     while (firstSource != right && firstSource != upStep)
360     {
361         upStep = firstSource;
362         firstSource = Links.Unsync.GetSource(upStep);
363     }
364     if (firstSource == right)
365     {
366         return handler(new LinkAddress<LinkIndex>(stepFrom));
367     }
368     return Constants.Continue;
369 }
370
371 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

362 private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↳ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
    ↳ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
    ↳ right));
363
364 [MethodImpl(MethodImplOptions.AggressiveInlining)]
365 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex stepFrom)
366 {
367     var upStep = stepFrom;
368     var firstTarget = Links.Unsync.GetSource(upStep);
369     while (firstTarget != left && firstTarget != upStep)
370     {
371         upStep = firstTarget;
372         firstTarget = Links.Unsync.GetTarget(upStep);
373     }
374     if (firstTarget == left)
375     {
376         return handler(new LinkAddress<LinkIndex>(stepFrom));
377     }
378     return Constants.Continue;
379 }
380
381 #endregion
382
383 #region Update
384
385 [MethodImpl(MethodImplOptions.AggressiveInlining)]
386 public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
387 {
388     var sequence = restrictions.SkipFirst();
389     var newSequence = substitution.SkipFirst();
390     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
391     {
392         return Constants.Null;
393     }
394     if (sequence.IsNullOrEmpty())
395     {
396         return Create(substitution);
397     }
398     if (newSequence.IsNullOrEmpty())
399     {
400         Delete(restrictions);
401         return Constants.Null;
402     }
403     return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
404     {
405         ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
406         Links.EnsureLinkExists(newSequence);
407         return UpdateCore(sequence, newSequence);
408     })));
409 }
410
411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
412 private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
413 {
414     LinkIndex bestVariant;
415     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
    ↳ !sequence.EqualTo(newSequence))
416     {
417         bestVariant = CompactCore(newSequence);
418     }
419     else
420     {
421         bestVariant = CreateCore(newSequence);
422     }
423     // TODO: Check all options only ones before loop execution
424     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
    ↳ маркером,
425     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
    ↳ можно получить имея только фактические последовательности.
426     foreach (var variant in Each(sequence))
427     {
428         if (variant != bestVariant)
429         {
430             UpdateOneCore(variant, bestVariant);
431         }
432     }
433     return bestVariant;

```

```

434 }
435
436 [MethodImpl(MethodImplOptions.AggressiveInlining)]
437 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
438 {
439     if (Options.UseGarbageCollection)
440     {
441         var sequenceElements = GetSequenceElements(sequence);
442         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
443         var sequenceLink = GetSequenceByElements(sequenceElements);
444         var newSequenceElements = GetSequenceElements(newSequence);
445         var newSequenceLink = GetSequenceByElements(newSequenceElements);
446         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
447         {
448             if (sequenceLink != Constants.Null)
449             {
450                 Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
451             }
452             Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
453         }
454         ClearGarbage(sequenceElementsContents.Source);
455         ClearGarbage(sequenceElementsContents.Target);
456     }
457     else
458     {
459         if (Options.UseSequenceMarker)
460         {
461             var sequenceElements = GetSequenceElements(sequence);
462             var sequenceLink = GetSequenceByElements(sequenceElements);
463             var newSequenceElements = GetSequenceElements(newSequence);
464             var newSequenceLink = GetSequenceByElements(newSequenceElements);
465             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
466             {
467                 if (sequenceLink != Constants.Null)
468                 {
469                     Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
470                 }
471                 Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
472             }
473         }
474         else
475         {
476             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
477             {
478                 Links.Unsync.MergeAndDelete(sequence, newSequence);
479             }
480         }
481     }
482 }
483
484 #endregion
485
486 #region Delete
487
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 public void Delete(IList<LinkIndex> restrictions)
490 {
491     _sync.ExecuteWriteOperation(() =>
492     {
493         var sequence = restrictions.SkipFirst();
494         // TODO: Check all options only ones before loop execution
495         foreach (var linkToDelete in Each(sequence))
496         {
497             DeleteOneCore(linkToDelete);
498         }
499     });
500 }
501
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 #region Garbage Collection
591
592 /// <remarks>
593 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
594   ↳ определить извне или в унаследованном классе
595 /// </remarks>
596 [MethodImpl(MethodImplOptions.AggressiveInlining)]
597 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
598   ↳ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
599
600 [MethodImpl(MethodImplOptions.AggressiveInlining)]
601 private void ClearGarbage(LinkIndex link)
602 {
603     if (IsGarbage(link))
604     {
605         var contents = new Link<ulong>(Links.GetLink(link));
606         Links.Unsync.Delete(link);
607         ClearGarbage(contents.Source);
608         ClearGarbage(contents.Target);
609     }
610 }
611
612 #endregion
613
614 #region Walkers
615
616 [MethodImpl(MethodImplOptions.AggressiveInlining)]
617 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
618 {
619     return _sync.ExecuteReadOperation(() =>
620     {
621         var links = Links.Unsync;
622         foreach (var part in Options.Walker.Walk(sequence))
623         {
624             if (!handler(part))
625             {
626                 return false;
627             }
628         }
629         return true;
630     });
631 }
632
633 public class Matcher : RightSequenceWalker<LinkIndex>
634 {
635     private readonly Sequences _sequences;
636     private readonly IList<LinkIndex> _patternSequence;
637     private readonly HashSet<LinkIndex> _linksInSequence;
638     private readonly HashSet<LinkIndex> _results;
639     private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
640     private readonly HashSet<LinkIndex> _readAsElements;
641     private int _filterPosition;
642
643     [MethodImpl(MethodImplOptions.AggressiveInlining)]
644     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
645         ↳ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
646         ↳ HashSet<LinkIndex> readAsElements = null)
647         : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
648     {
649         _sequences = sequences;
650         _patternSequence = patternSequence;
651         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
652             ↳ Links.Constants.Any && x != ZeroOrMany));
653         _results = results;
654         _stopableHandler = stopableHandler;
655         _readAsElements = readAsElements;
656     }
657
658     [MethodImpl(MethodImplOptions.AggressiveInlining)]
659     protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
660     ↳ (_readAsElements != null && _readAsElements.Contains(link)) ||
661     ↳ _linksInSequence.Contains(link);
662
663     [MethodImpl(MethodImplOptions.AggressiveInlining)]
664     public bool FullMatch(LinkIndex sequenceToMatch)
665     {
666         _filterPosition = 0;
667         foreach (var part in Walk(sequenceToMatch))
668         {
669             if (!FullMatchCore(part))
670             {
671                 return false;
672             }
673         }
674         return true;
675     }
676
677     private bool FullMatchCore(LinkIndex part)
678     {
679         if (part == Options.SequenceMarkerLink)
680             return true;
681
682         if (part == ZeroOrMany)
683             return false;
684
685         if (part == Links.Constants.Any)
686             return true;
687
688         if (part == Links.Constants.Nothing)
689             return false;
690
691         if (part == Links.Constants.Null)
692             return false;
693
694         if (part == Links.Constants.Empty)
695             return false;
696
697         if (part == Links.Constants.False)
698             return false;
699
700         if (part == Links.Constants.True)
701             return false;
702
703         if (part == Links.Constants.Default)
704             return false;
705
706         if (part == Links.Constants.Min)
707             return false;
708
709         if (part == Links.Constants.Max)
710             return false;
711
712         if (part == Links.Constants.First)
713             return false;
714
715         if (part == Links.Constants.Last)
716             return false;
717
718         if (part == Links.Constants.All)
719             return false;
720
721         if (part == Links.Constants.None)
722             return false;
723
724         if (part == Links.Constants.AllOf)
725             return false;
726
727         if (part == Links.Constants.AnyOf)
728             return false;
729
730         if (part == Links.Constants.NoneOf)
731             return false;
732
733         if (part == Links.Constants.AllOrNone)
734             return false;
735
736         if (part == Links.Constants.AnyOrNone)
737             return false;
738
739         if (part == Links.Constants.AllOrAny)
740             return false;
741
742         if (part == Links.Constants.AnyOrAll)
743             return false;
744
745         if (part == Links.Constants.AllOrAnyOf)
746             return false;
747
748         if (part == Links.Constants.AnyOrAllOf)
749             return false;
750
751         if (part == Links.Constants.AllOrNoneOf)
752             return false;
753
754         if (part == Links.Constants.AnyOrNoneOf)
755             return false;
756
757         if (part == Links.Constants.AllOrAnyOf)
758             return false;
759
760         if (part == Links.Constants.AnyOrAllOf)
761             return false;
762
763         if (part == Links.Constants.AllOrNoneOf)
764             return false;
765
766         if (part == Links.Constants.AnyOrNoneOf)
767             return false;
768
769         if (part == Links.Constants.AllOrAnyOf)
770             return false;
771
772         if (part == Links.Constants.AnyOrAllOf)
773             return false;
774
775         if (part == Links.Constants.AllOrNoneOf)
776             return false;
777
778         if (part == Links.Constants.AnyOrNoneOf)
779             return false;
780
781         if (part == Links.Constants.AllOrAnyOf)
782             return false;
783
784         if (part == Links.Constants.AnyOrAllOf)
785             return false;
786
787         if (part == Links.Constants.AllOrNoneOf)
788             return false;
789
790         if (part == Links.Constants.AnyOrNoneOf)
791             return false;
792
793         if (part == Links.Constants.AllOrAnyOf)
794             return false;
795
796         if (part == Links.Constants.AnyOrAllOf)
797             return false;
798
799         if (part == Links.Constants.AllOrNoneOf)
800             return false;
801
802         if (part == Links.Constants.AnyOrNoneOf)
803             return false;
804
805         if (part == Links.Constants.AllOrAnyOf)
806             return false;
807
808         if (part == Links.Constants.AnyOrAllOf)
809             return false;
810
811         if (part == Links.Constants.AllOrNoneOf)
812             return false;
813
814         if (part == Links.Constants.AnyOrNoneOf)
815             return false;
816
817         if (part == Links.Constants.AllOrAnyOf)
818             return false;
819
820         if (part == Links.Constants.AnyOrAllOf)
821             return false;
822
823         if (part == Links.Constants.AllOrNoneOf)
824             return false;
825
826         if (part == Links.Constants.AnyOrNoneOf)
827             return false;
828
829         if (part == Links.Constants.AllOrAnyOf)
830             return false;
831
832         if (part == Links.Constants.AnyOrAllOf)
833             return false;
834
835         if (part == Links.Constants.AllOrNoneOf)
836             return false;
837
838         if (part == Links.Constants.AnyOrNoneOf)
839             return false;
840
841         if (part == Links.Constants.AllOrAnyOf)
842             return false;
843
844         if (part == Links.Constants.AnyOrAllOf)
845             return false;
846
847         if (part == Links.Constants.AllOrNoneOf)
848             return false;
849
850         if (part == Links.Constants.AnyOrNoneOf)
851             return false;
852
853         if (part == Links.Constants.AllOrAnyOf)
854             return false;
855
856         if (part == Links.Constants.AnyOrAllOf)
857             return false;
858
859         if (part == Links.Constants.AllOrNoneOf)
860             return false;
861
862         if (part == Links.Constants.AnyOrNoneOf)
863             return false;
864
865         if (part == Links.Constants.AllOrAnyOf)
866             return false;
867
868         if (part == Links.Constants.AnyOrAllOf)
869             return false;
870
871         if (part == Links.Constants.AllOrNoneOf)
872             return false;
873
874         if (part == Links.Constants.AnyOrNoneOf)
875             return false;
876
877         if (part == Links.Constants.AllOrAnyOf)
878             return false;
879
880         if (part == Links.Constants.AnyOrAllOf)
881             return false;
882
883         if (part == Links.Constants.AllOrNoneOf)
884             return false;
885
886         if (part == Links.Constants.AnyOrNoneOf)
887             return false;
888
889         if (part == Links.Constants.AllOrAnyOf)
890             return false;
891
892         if (part == Links.Constants.AnyOrAllOf)
893             return false;
894
895         if (part == Links.Constants.AllOrNoneOf)
896             return false;
897
898         if (part == Links.Constants.AnyOrNoneOf)
899             return false;
900
901         if (part == Links.Constants.AllOrAnyOf)
902             return false;
903
904         if (part == Links.Constants.AnyOrAllOf)
905             return false;
906
907         if (part == Links.Constants.AllOrNoneOf)
908             return false;
909
910         if (part == Links.Constants.AnyOrNoneOf)
911             return false;
912
913         if (part == Links.Constants.AllOrAnyOf)
914             return false;
915
916         if (part == Links.Constants.AnyOrAllOf)
917             return false;
918
919         if (part == Links.Constants.AllOrNoneOf)
920             return false;
921
922         if (part == Links.Constants.AnyOrNoneOf)
923             return false;
924
925         if (part == Links.Constants.AllOrAnyOf)
926             return false;
927
928         if (part == Links.Constants.AnyOrAllOf)
929             return false;
930
931         if (part == Links.Constants.AllOrNoneOf)
932             return false;
933
934         if (part == Links.Constants.AnyOrNoneOf)
935             return false;
936
937         if (part == Links.Constants.AllOrAnyOf)
938             return false;
939
940         if (part == Links.Constants.AnyOrAllOf)
941             return false;
942
943         if (part == Links.Constants.AllOrNoneOf)
944             return false;
945
946         if (part == Links.Constants.AnyOrNoneOf)
947             return false;
948
949         if (part == Links.Constants.AllOrAnyOf)
950             return false;
951
952         if (part == Links.Constants.AnyOrAllOf)
953             return false;
954
955         if (part == Links.Constants.AllOrNoneOf)
956             return false;
957
958         if (part == Links.Constants.AnyOrNoneOf)
959             return false;
960
961         if (part == Links.Constants.AllOrAnyOf)
962             return false;
963
964         if (part == Links.Constants.AnyOrAllOf)
965             return false;
966
967         if (part == Links.Constants.AllOrNoneOf)
968             return false;
969
970         if (part == Links.Constants.AnyOrNoneOf)
971             return false;
972
973         if (part == Links.Constants.AllOrAnyOf)
974             return false;
975
976         if (part == Links.Constants.AnyOrAllOf)
977             return false;
978
979         if (part == Links.Constants.AllOrNoneOf)
980             return false;
981
982         if (part == Links.Constants.AnyOrNoneOf)
983             return false;
984
985         if (part == Links.Constants.AllOrAnyOf)
986             return false;
987
988         if (part == Links.Constants.AnyOrAllOf)
989             return false;
990
991         if (part == Links.Constants.AllOrNoneOf)
992             return false;
993
994         if (part == Links.Constants.AnyOrNoneOf)
995             return false;
996
997         if (part == Links.Constants.AllOrAnyOf)
998             return false;
999
1000        if (part == Links.Constants.AnyOrAllOf)
1001            return false;
1002
1003        if (part == Links.Constants.AllOrNoneOf)
1004            return false;
1005
1006        if (part == Links.Constants.AnyOrNoneOf)
1007            return false;
1008
1009        if (part == Links.Constants.AllOrAnyOf)
1010            return false;
1011
1012        if (part == Links.Constants.AnyOrAllOf)
1013            return false;
1014
1015        if (part == Links.Constants.AllOrNoneOf)
1016            return false;
1017
1018        if (part == Links.Constants.AnyOrNoneOf)
1019            return false;
1020
1021        if (part == Links.Constants.AllOrAnyOf)
1022            return false;
1023
1024        if (part == Links.Constants.AnyOrAllOf)
1025            return false;
1026
1027        if (part == Links.Constants.AllOrNoneOf)
1028            return false;
1029
1030        if (part == Links.Constants.AnyOrNoneOf)
1031            return false;
1032
1033        if (part == Links.Constants.AllOrAnyOf)
1034            return false;
1035
1036        if (part == Links.Constants.AnyOrAllOf)
1037            return false;
1038
1039        if (part == Links.Constants.AllOrNoneOf)
1040            return false;
1041
1042        if (part == Links.Constants.AnyOrNoneOf)
1043            return false;
1044
1045        if (part == Links.Constants.AllOrAnyOf)
1046            return false;
1047
1048        if (part == Links.Constants.AnyOrAllOf)
1049            return false;
1050
1051        if (part == Links.Constants.AllOrNoneOf)
1052            return false;
1053
1054        if (part == Links.Constants.AnyOrNoneOf)
1055            return false;
1056
1057        if (part == Links.Constants.AllOrAnyOf)
1058            return false;
1059
1060        if (part == Links.Constants.AnyOrAllOf)
1061            return false;
1062
1063        if (part == Links.Constants.AllOrNoneOf)
1064            return false;
1065
1066        if (part == Links.Constants.AnyOrNoneOf)
1067            return false;
1068
1069        if (part == Links.Constants.AllOrAnyOf)
1070            return false;
1071
1072        if (part == Links.Constants.AnyOrAllOf)
1073            return false;
1074
1075        if (part == Links.Constants.AllOrNoneOf)
1076            return false;
1077
1078        if (part == Links.Constants.AnyOrNoneOf)
1079            return false;
1080
1081        if (part == Links.Constants.AllOrAnyOf)
1082            return false;
1083
1084        if (part == Links.Constants.AnyOrAllOf)
1085            return false;
1086
1087        if (part == Links.Constants.AllOrNoneOf)
1088            return false;
1089
1090        if (part == Links.Constants.AnyOrNoneOf)
1091            return false;
1092
1093        if (part == Links.Constants.AllOrAnyOf)
1094            return false;
1095
1096        if (part == Links.Constants.AnyOrAllOf)
1097            return false;
1098
1099        if (part == Links.Constants.AllOrNoneOf)
1100            return false;
1101
1102        if (part == Links.Constants.AnyOrNoneOf)
1103            return false;
1104
1105        if (part == Links.Constants.AllOrAnyOf)
1106            return false;
1107
1108        if (part == Links.Constants.AnyOrAllOf)
1109            return false;
1110
1111        if (part == Links.Constants.AllOrNoneOf)
1112            return false;
1113
1114        if (part == Links.Constants.AnyOrNoneOf)
1115            return false;
1116
1117        if (part == Links.Constants.AllOrAnyOf)
1118            return false;
1119
1120        if (part == Links.Constants.AnyOrAllOf)
1121            return false;
1122
1123        if (part == Links.Constants.AllOrNoneOf)
1124            return false;
1125
1126        if (part == Links.Constants.AnyOrNoneOf)
1127            return false;
1128
1129        if (part == Links.Constants.AllOrAnyOf)
1130            return false;
1131
1132        if (part == Links.Constants.AnyOrAllOf)
1133            return false;
1134
1135        if (part == Links.Constants.AllOrNoneOf)
1136            return false;
1137
1138        if (part == Links.Constants.AnyOrNoneOf)
1139            return false;
1140
1141        if (part == Links.Constants.AllOrAnyOf)
1142            return false;
1143
1144        if (part == Links.Constants.AnyOrAllOf)
1145            return false;
1146
1147        if (part == Links.Constants.AllOrNoneOf)
1148            return false;
1149
1150        if (part == Links.Constants.AnyOrNoneOf)
1151            return false;
1152
1153        if (part == Links.Constants.AllOrAnyOf)
1154            return false;
1155
1156        if (part == Links.Constants.AnyOrAllOf)
1157            return false;
1158
1159        if (part == Links.Constants.AllOrNoneOf)
1160            return false;
1161
1162        if (part == Links.Constants.AnyOrNoneOf)
1163            return false;
1164
1165        if (part == Links.Constants.AllOrAnyOf)
1166            return false;
1167
1168        if (part == Links.Constants.AnyOrAllOf)
1169            return false;
1170
1171        if (part == Links.Constants.AllOrNoneOf)
1172            return false;
1173
1174        if (part == Links.Constants.AnyOrNoneOf)
1175            return false;
1176
1177        if (part == Links.Constants.AllOrAnyOf)
1178            return false;
1179
1180        if (part == Links.Constants.AnyOrAllOf)
1181            return false;
1182
1183        if (part == Links.Constants.AllOrNoneOf)
1184            return false;
1185
1186        if (part == Links.Constants.AnyOrNoneOf)
1187            return false;
1188
1189        if (part == Links.Constants.AllOrAnyOf)
1190            return false;
1191
1192        if (part == Links.Constants.AnyOrAllOf)
1193            return false;
1194
1195        if (part == Links.Constants.AllOrNoneOf)
1196            return false;
1197
1198        if (part == Links.Constants.AnyOrNoneOf)
1199            return false;
1200
1201        if (part == Links.Constants.AllOrAnyOf)
1202            return false;
1203
1204        if (part == Links.Constants.AnyOrAllOf)
1205            return false;
1206
1207        if (part == Links.Constants.AllOrNoneOf)
1208            return false;
1209
1210        if (part == Links.Constants.AnyOrNoneOf)
1211            return false;
1212
1213        if (part == Links.Constants.AllOrAnyOf)
1214            return false;
1215
1216        if (part == Links.Constants.AnyOrAllOf)
1217            return false;
1218
1219        if (part == Links.Constants.AllOrNoneOf)
1220            return false;
1221
1222        if (part == Links.Constants.AnyOrNoneOf)
1223            return false;
1224
1225        if (part == Links.Constants.AllOrAnyOf)
1226            return false;
1227
1228        if (part == Links.Constants.AnyOrAllOf)
1229            return false;
1230
1231        if (part == Links.Constants.AllOrNoneOf)
1232            return false;
1233
1234        if (part == Links.Constants.AnyOrNoneOf)
1235            return false;
1236
1237        if (part == Links.Constants.AllOrAnyOf)
1238            return false;
1239
1240        if (part == Links.Constants.AnyOrAllOf)
1241            return false;
1242
1243        if (part == Links.Constants.AllOrNoneOf)
1244            return false;
1245
1246        if (part == Links.Constants.AnyOrNoneOf)
1247            return false;
1248
1249        if (part == Links.Constants.AllOrAnyOf)
1250            return false;
1251
1252        if (part == Links.Constants.AnyOrAllOf)
1253            return false;
1254
1255        if (part == Links.Constants.AllOrNoneOf)
1256            return false;
1257
1258        if (part == Links.Constants.AnyOrNoneOf)
1259            return false;
1260
1261        if (part == Links.Constants.AllOrAnyOf)
1262            return false;
1263
1264        if (part == Links.Constants.AnyOrAllOf)
1265            return false;
1266
1267        if (part == Links.Constants.AllOrNoneOf)
1268            return false;
1269
1270        if (part == Links.Constants.AnyOrNoneOf)
1271            return false;
1272
1273        if (part == Links.Constants.AllOrAnyOf)
1274            return false;
1275
1276        if (part == Links.Constants.AnyOrAllOf)
1277            return false;
1278
1279        if (part == Links.Constants.AllOrNoneOf)
1280            return false;
1281
1282        if (part == Links.Constants.AnyOrNoneOf)
1283            return false;
1284
1285        if (part == Links.Constants.AllOrAnyOf)
1286            return false;
1287
1288        if (part == Links.Constants.AnyOrAllOf)
1289            return false;
1290
1291        if (part == Links.Constants.AllOrNoneOf)
1292            return false;
1293
1294        if (part == Links.Constants.AnyOrNoneOf)
1295            return false;
1296
1297        if (part == Links.Constants.AllOrAnyOf)
1298            return false;
1299
1300        if (part == Links.Constants.AnyOrAllOf)
1301            return false;
1302
1303        if (part == Links.Constants.AllOrNoneOf)
1304            return false;
1305
1306        if (part == Links.Constants.AnyOrNoneOf)
1307            return false;
1308
1309        if (part == Links.Constants.AllOrAnyOf)
1310            return false;
1311
1312        if (part == Links.Constants.AnyOrAllOf)
1313            return false;
1314
1315        if (part == Links.Constants.AllOrNoneOf)
1316            return false;
1317
1318        if (part == Links.Constants.AnyOrNoneOf)
1319            return false;
1320
1321        if (part == Links.Constants.AllOrAnyOf)
1322            return false;
1323
1324        if (part == Links.Constants.AnyOrAllOf)
1325            return false;
1326
1327        if (part == Links.Constants.AllOrNoneOf)
1328            return false;
1329
1330        if (part == Links.Constants.AnyOrNoneOf)
1331            return false;
1332
1333        if (part == Links.Constants.AllOrAnyOf)
1334            return false;
1335
1336        if (part == Links.Constants.AnyOrAllOf)
1337            return false;
1338
1339        if (part == Links.Constants.AllOrNoneOf)
1340            return false;
1341
1342        if (part == Links.Constants.AnyOrNoneOf)
1343            return false;
1344
1345        if (part == Links.Constants.AllOrAnyOf)
1346            return false;
1347
1348        if (part == Links.Constants.AnyOrAllOf)
1349            return false;
1350
1351        if (part == Links.Constants.AllOrNoneOf)
1352            return false;
1353
1354        if (part == Links.Constants.AnyOrNoneOf)
1355            return false;
1356
1357        if (part == Links.Constants.AllOrAnyOf)
1358            return false;
1359
1360        if (part == Links.Constants.AnyOrAllOf)
1361            return false;
1362
1363        if (part == Links.Constants.AllOrNoneOf)
1364            return false;
1365
1366        if (part == Links.Constants.AnyOrNoneOf)
1367            return false;
1368
1369        if (part == Links.Constants.AllOrAnyOf)
1370            return false;
1371
1372        if (part == Links.Constants.AnyOrAllOf)
1373            return false;
1374
1375        if (part == Links.Constants.AllOrNoneOf)
1376            return false;
1377
1378        if (part == Links.Constants.AnyOrNoneOf)
1379            return false;
1380
1381        if (part == Links.Constants.AllOrAnyOf)
1382            return false;
1383
1384        if (part == Links.Constants.AnyOrAllOf)
1385            return false;
1386
1387        if (part == Links.Constants.AllOrNoneOf)
1388            return false;
1389
1390        if (part == Links.Constants.AnyOrNoneOf)
1391            return false;
1392
1393        if (part == Links.Constants.AllOrAnyOf)
1394            return false;
1395
1396        if (part == Links.Constants.AnyOrAllOf)
1397            return false;
1398
1399        if (part == Links.Constants.AllOrNoneOf)
1400            return false;
1401
1402        if (part == Links.Constants.AnyOrNoneOf)
1403            return false;
1404
1405        if (part == Links.Constants.AllOrAnyOf)
1406            return false;
1407
1408        if (part == Links.Constants.AnyOrAllOf)
1409            return false;
1410
1411        if (part == Links.Constants.AllOrNoneOf)
1412            return false;
1413
1414        if (part == Links.Constants.AnyOrNoneOf)
1415            return false;
1416
1417        if (part == Links.Constants.AllOrAnyOf)
1418            return false;
1419
1420        if (part == Links.Constants.AnyOrAllOf)
1421            return false;
1422
1423        if (part == Links.Constants.AllOrNoneOf)
1424            return false;
1425
1426        if (part == Links.Constants.AnyOrNoneOf)
1427            return false;
1428
1429        if (part == Links.Constants.AllOrAnyOf)
1430            return false;
1431
1432        if (part == Links.Constants.AnyOrAllOf)
1433            return false;
1434
1435        if (part == Links.Constants.AllOrNoneOf)
1436            return false;
1437
1438        if (part == Links.Constants.AnyOrNoneOf)
1439            return false;
1440
1441        if (part == Links.Constants.AllOrAnyOf)
1442            return false;
1443
1444        if (part == Links.Constants.AnyOrAllOf)
1445            return false;
1446
1447        if (part == Links.Constants.AllOrNoneOf)
1448            return false;
1449
1450        if (part == Links.Constants.AnyOrNoneOf)
1451            return false;
1452
1453        if (part == Links.Constants.AllOrAnyOf)
1454            return false;
1455
1456        if (part == Links.Constants.AnyOrAllOf)
1457            return false;
1458
1459        if (part == Links.Constants.AllOrNoneOf)
1460            return false;
1461
1462        if (part == Links.Constants.AnyOrNoneOf)
1463            return false;
1464
1465        if (part == Links.Constants.AllOrAnyOf)
1466            return false;
1467
1468        if (part == Links.Constants.AnyOrAllOf)
1469            return false;
1470
1471        if (part == Links.Constants.AllOrNoneOf)
1472            return false;
1473
1474        if (part == Links.Constants.AnyOrNoneOf)
1475            return false;
1476
1477        if (part == Links.Constants.AllOrAnyOf)
1478            return false;
1479
1480        if (part == Links.Constants.AnyOrAllOf)
1481            return false;
1482
1483        if (part == Links.Constants.AllOrNoneOf)
1484            return false;
1485
1486        if (part == Links.Constants.AnyOrNoneOf)
1487            return false;
1488
1489        if (part == Links.Constants.AllOrAnyOf)
1490            return false;
1491
1492        if (part == Links.Constants.AnyOrAllOf)
1493            return false;
1494
1495        if (part == Links.Constants.AllOrNoneOf)
1496            return false;
1497
1498        if (part == Links.Constants.AnyOrNoneOf)
1499            return false;
1500
1501        if (part == Links.Constants.AllOrAnyOf)
1502            return false;
1503
1504        if (part == Links.Constants.AnyOrAllOf)
1505            return false;
1506
1507        if (part == Links.Constants.AllOrNoneOf)
1508            return false;
1509
1510        if (part == Links.Constants.AnyOrNoneOf)
1511            return false;
1512
1513        if (part == Links.Constants.AllOrAnyOf)
1514            return false;
1515
1516        if (part == Links.Constants.AnyOrAllOf)
1517            return false;
1518
1519        if (part == Links.Constants.AllOrNoneOf)
1520            return false;
1521
1522        if (part == Links.Constants.AnyOrNoneOf)
1523            return false;
1524
1525        if (part == Links.Constants.AllOrAnyOf)
1526            return false;
1527
1528        if (part == Links.Constants.AnyOrAllOf)
1529            return false;
1530
1531        if (part == Links.Constants.AllOrNoneOf)
1532            return false;
1533
1534        if (part == Links.Constants.AnyOrNoneOf)
1535            return false;
1536
1537        if (part == Links.Constants.AllOrAnyOf)
1538            return false;
1539
1540        if (part == Links.Constants.AnyOrAllOf)
1541            return false;
1542
1543        if (part == Links.Constants.AllOrNoneOf)
1544            return false;
1545
1546        if (part == Links.Constants.AnyOrNoneOf)
1547            return false;
1548
1549        if (part == Links.Constants.AllOrAnyOf)
1550            return false;
1551
1552        if (part == Links.Constants.AnyOrAllOf)
1553            return false;
1554
1555        if (part == Links.Constants.AllOrNoneOf)
1556            return false;
1557
1558        if (part == Links.Constants.AnyOrNoneOf)
1559            return false;
1560
1561        if (part == Links.Constants.AllOrAnyOf)
1562            return false;
1563
1564        if (part == Links.Constants.AnyOrAllOf)
1565            return false;
1566
1567        if (part == Links.Constants.AllOrNoneOf)
1568            return false;
1569
1570        if (part == Links.Constants.AnyOrNoneOf)
1571            return false;
1572
1573        if (part == Links.Constants.AllOrAnyOf)
1574            return false;
1575
1576        if (part == Links.Constants.AnyOrAllOf)
1577            return false;
1578
1579        if (part == Links.Constants.AllOrNoneOf)
1580            return false;
1581
1582        if (part == Links.Constants.AnyOrNoneOf)
1583            return false;
1584
1585        if (part == Links.Constants.AllOrAnyOf)
1586            return false;
1587
1588        if (part == Links.Constants.AnyOrAllOf)
1589            return false;
1590
1591        if (part == Links.Constants.AllOrNoneOf)
1592            return false;
1593
1594        if (part == Links.Constants.AnyOrNoneOf)
1595            return false;
1596
1597        if (part == Links.Constants.AllOrAnyOf)
1598            return false;
1599
1600        if (part == Links.Constants.AnyOrAllOf)
1601            return false;
1602
1603        if (part == Links.Constants.AllOrNoneOf)
1604            return false;
1605
1606        if (part == Links.Constants.AnyOrNoneOf)
1607            return false;
1608
1609        if (part == Links.Constants.AllOrAnyOf)
1610            return false;
1611
1612        if (part == Links.Constants.AnyOrAllOf)
1613            return false;
1614
1615        if (part == Links.Constants.AllOrNoneOf)
1616            return false;
1617
1618        if (part == Links.Constants.AnyOrNoneOf)
1619            return false;
1620
1621        if (part == Links.Constants.AllOrAnyOf)
1622            return false;
1623
1624        if (part == Links.Constants.AnyOrAllOf)
1625            return false;
1626
1627        if (part == Links.Constants.AllOrNoneOf)
1628            return false;
1629
1630        if (part == Links.Constants.AnyOrNoneOf)
1631            return false;
1632
1633        if (part == Links.Constants.AllOrAnyOf)
1634            return false;
1635
1636        if (part == Links.Constants.AnyOrAllOf)
1637            return false;
1638
1639        if (part == Links.Constants.AllOrNoneOf)
1640            return false;
1641
1642        if (part == Links.Constants.AnyOrNoneOf)
1643            return false;
1644
1645        if (part == Links.Constants.AllOrAnyOf)
1646            return false;
1647
1648        if (part == Links.Constants.AnyOrAllOf)
1649            return false;
1650
1651        if (part == Links.Constants.AllOrNoneOf)
1652            return false;
1653
1654        if (part == Links.Constants.AnyOrNoneOf)
1655            return false;
1656
1657        if (part == Links.Constants.AllOrAnyOf)
1658            return false;
1659
1660        if (part == Links.Constants.AnyOrAllOf)
1661            return false;
1662
1663        if (part == Links.Constants.AllOrNoneOf)
1664            return false;
1665
1666        if (part == Links.Constants.AnyOrNoneOf)
1667            return false;
1668
1669        if (part == Links.Constants.AllOrAnyOf)
1670            return false;
1671
1672        if (part == Links.Constants.AnyOrAllOf)
1673            return false;
1674
1675        if (part == Links.Constants.AllOrNoneOf)
1676            return false;
1677
1678        if (part == Links.Constants.AnyOrNoneOf)
1679            return false;
1680
1681        if (part == Links.Constants.AllOrAnyOf)
1682            return false;
1683
1684        if (part == Links.Constants.AnyOrAllOf)
1685            return false;
1686
1687        if (part == Links.Constants.AllOrNoneOf)
1688            return false;
1689
1690        if (part == Links.Constants.AnyOrNoneOf)
1691            return false;
1692
1693        if (part == Links.Constants.AllOrAnyOf)
1694            return false;
1695
1696        if (part == Links.Constants.AnyOrAllOf)
1697            return false;
1698
1699        if (part == Links.Constants.AllOrNoneOf)
1700            return false;
1701
1702        if (part == Links.Constants.AnyOrNoneOf)
1703            return false;
1704
1705        if (part == Links.Constants.AllOrAnyOf)
1706            return false;
1707
1708        if (part == Links.Constants.AnyOrAllOf)
1709            return false;
1710
1711        if (part == Links.Constants.AllOrNoneOf)
1712            return false;
1713
1714        if (part == Links.Constants.AnyOrNoneOf)
1715            return false;
1716
1717        if (part == Links.Constants.AllOrAnyOf)
1718            return false;
1719
1720        if (part == Links.Constants.AnyOrAllOf)
1721            return false;
1722
1723        if (part == Links.Constants.AllOrNoneOf)
1724            return false;
1725
1726        if (part == Links.Constants.AnyOrNoneOf)
1727            return false;
1728
1729        if (part == Links.Constants.AllOrAnyOf)
1730            return false;
1731
1732        if (part == Links.Constants.AnyOrAllOf)
1733            return false;
1734
1735        if (part == Links.Constants.AllOrNoneOf)
1736            return false;
1737
1738        if (part == Links.Constants.AnyOrNoneOf)
1739            return false;
1740
1741        if (part == Links.Constants.AllOrAnyOf)
1742            return false;
1743
1744        if (part == Links.Constants.AnyOrAllOf)
1745            return false;
1746
1747        if (part == Links.Constants.AllOrNoneOf)
1748            return false;
1749
1750        if (part == Links.Constants.AnyOrNoneOf)
1751            return false;
1752
1753        if (part == Links.Constants.AllOrAnyOf)
1754            return false;
1755
1756        if (part == Links.Constants.AnyOrAllOf)
1757            return false;
1758
1759        if (part == Links.Constants.AllOrNoneOf)
1760            return false;
1761
1762        if (part == Links.Constants.AnyOrNoneOf)
1763            return false;
1764
1765        if (part == Links.Constants.AllOrAnyOf)
1766            return false;
1767
1768        if (part == Links.Constants.AnyOrAllOf)
1769            return false;
1770
1771        if (part == Links.Constants.AllOrNoneOf)
1772            return false;
1773
1774        if (part == Links.Constants.AnyOrNoneOf)
1775            return false;
1776
1777        if (part == Links.Constants.AllOrAnyOf)
1778            return false;
1779
1780        if (part == Links.Constants.AnyOrAllOf)
1781            return false;
1782
1783        if (part == Links.Constants.AllOrNoneOf)
1784            return false;
1785
1786        if (part == Links.Constants.AnyOrNoneOf)
1787            return false;
1788
1789        if (part == Links.Constants.AllOrAnyOf)
1790            return false;
1791
1792        if (part == Links.Constants.AnyOrAllOf)
1793            return false;
1794
1795        if (part == Links.Constants.AllOrNoneOf)
1796            return false;
1797
1798        if (part == Links.Constants.AnyOrNoneOf)
1799            return false;
1800
1801        if (part == Links.Constants.AllOrAnyOf)
1802            return false;
1803
1804        if (part == Links.Constants.AnyOrAllOf)
1805            return false;
1806
1807        if (part == Links.Constants.AllOrNoneOf)
1808            return false;
1809
1810        if (part == Links.Constants
```

```

663         {
664             break;
665         }
666     }
667     return _filterPosition == _patternSequence.Count;
668 }
669
670 [MethodImpl(MethodImplOptions.AggressiveInlining)]
671 private bool FullMatchCore(LinkIndex element)
672 {
673     if (_filterPosition == _patternSequence.Count)
674     {
675         _filterPosition = -2; // Длиннее чем нужно
676         return false;
677     }
678     if (_patternSequence[_filterPosition] != Links.Constants.Any
679         && element != _patternSequence[_filterPosition])
680     {
681         _filterPosition = -1;
682         return false; // Начинается/Продолжается иначе
683     }
684     _filterPosition++;
685     return true;
686 }
687
688 [MethodImpl(MethodImplOptions.AggressiveInlining)]
689 public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
690 {
691     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
692     if (FullMatch(sequenceToMatch))
693     {
694         _results.Add(sequenceToMatch);
695     }
696 }
697
698 [MethodImpl(MethodImplOptions.AggressiveInlining)]
699 public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
700 {
701     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
702     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
703     {
704         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
705     }
706     return Links.Constants.Continue;
707 }
708
709 [MethodImpl(MethodImplOptions.AggressiveInlining)]
710 public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
711 {
712     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
713     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
714     if (sequence != Links.Constants.Null && FullMatch(sequenceToMatch) &&
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 {

```

```

741     if (_filterPosition == (_patternSequence.Count - 1))
742     {
743         return false; // Нашлось
744     }
745     if (_filterPosition >= 0)
746     {
747         if (element == _patternSequence[_filterPosition + 1])
748         {
749             _filterPosition++;
750         }
751         else
752         {
753             _filterPosition = -1;
754         }
755     }
756     if (_filterPosition < 0)
757     {
758         if (element == _patternSequence[0])
759         {
760             _filterPosition = 0;
761         }
762     }
763     return true; // Ищем дальше
764 }
765
766 [MethodImpl(MethodImplOptions.AggressiveInlining)]
767 public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
768 {
769     if (PartialMatch(sequenceToMatch))
770     {
771         _results.Add(sequenceToMatch);
772     }
773 }
774
775 [MethodImpl(MethodImplOptions.AggressiveInlining)]
776 public LinkIndex HandlePartialMatched(ICollection<LinkIndex> restrictions)
777 {
778     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
779     if (PartialMatch(sequenceToMatch))
780     {
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 ./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 ./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
19     ↪ ILinks<TLink> must contain GetConstants function.
20     {
21         private static readonly EqualityComparer<TLink> _equalityComparer =
22         ↪ EqualityComparer<TLink>.Default;
23
24         public TLink SequenceMarkerLink
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29             set;
30         }
31
32         public bool UseCascadeUpdate
33         {
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             get;
36             [MethodImpl(MethodImplOptions.AggressiveInlining)]
37             set;
38         }
39
40         public bool UseCascadeDelete
41         {
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             get;
44             [MethodImpl(MethodImplOptions.AggressiveInlining)]
45             set;
46         }
47     }
48 }

```

```

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 public bool ReadFullSequence
126 {
127     [MethodImpl(MethodImplOptions.AggressiveInlining)]
128     get;
129     [MethodImpl(MethodImplOptions.AggressiveInlining)]
130     set;
131 }
132
133 // TODO: Реализовать компактификацию при чтении
134 //public bool EnforceSingleSequenceVersionOnRead { get; set; }
135 //public bool UseRequestMarker { get; set; }
136 //public bool StoreRequestResults { get; set; }
137
138 [MethodImpl(MethodImplOptions.AggressiveInlining)]
139 public void InitOptions(ISynchronizedLinks<TLink> links)
140 {
141     if (UseSequenceMarker)
142     {
143         if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
144         {
145             SequenceMarkerLink = links.CreatePoint();
146         }
147         else
148         {
149             if (!links.Exists(SequenceMarkerLink))
150             {
151                 var link = links.CreatePoint();
152                 if (!_equalityComparer.Equals(link, SequenceMarkerLink))
153                 {
154                     throw new InvalidOperationException("Cannot recreate sequence marker
155                     ↪ link.");
156                 }
157             }
158         }
159         if (MarkedSequenceMatcher == null)
160         {
161             MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
162             ↪ SequenceMarkerLink);
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         {

```

```

196         Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
197     }
198 }
199
200 [MethodImpl(MethodImplOptions.AggressiveInlining)]
201 public void ValidateOptions()
202 {
203     if (UseGarbageCollection && !UseSequenceMarker)
204     {
205         throw new NotSupportedException("To use garbage collection UseSequenceMarker
206             ↪ option must be on.");
207     }
208 }
209 }

```

1.85 ./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 ./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 parts = Links.GetLink(element);
32             var start = Links.Constants.IndexPart + 1;
33             for (var i = parts.Count - 1; i >= start; i--)
34             {
35                 var part = parts[i];
36                 if (IsElement(part))
37                 {
38                     yield return part;
39                 }
40             }
41         }
42     }
43 }

```

1.87 ./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;
43             do
44             {
45                 length *= 2;
46 #if USEARRAYPOOL
47                 var nextArray = ArrayPool.Allocate<ulong>(length);
48 #else
49                 var nextArray = new TLink[length];
50 #endif
51                 hasElements = false;
52                 for (var i = 0; i < array.Length; i++)
53                 {
54                     var candidate = array[i];
55                     if (_equalityComparer.Equals(array[i], default))
56                     {
57                         continue;
58                     }
59                     var doubletOffset = i * 2;
60                     if (_isElement(candidate))
61                     {
62                         nextArray[doubletOffset] = candidate;
63                     }
64                     else
65                     {
66                         var link = Links.GetLink(candidate);
67                         var linkSource = Links.GetSource(link);
68                         var linkTarget = Links.GetTarget(link);
69                         nextArray[doubletOffset] = linkSource;
70                         nextArray[doubletOffset + 1] = linkTarget;
71                         if (!hasElements)
72                         {
73                             hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
74                         }
75                     }
76                 }
77             } while (length < array.Length);
78             return array;
79         }
80     }
81 #if USEARRAYPOOL
82     if (array.Length > 1)
83     {
84

```

```

77         ArrayPool.Free(array);
78     }
79 #endif
80     array = nextArray;
81 }
82 while (hasElements);
83 var filledElementsCount = CountFilledElements(array);
84 if (filledElementsCount == array.Length)
85 {
86     return array;
87 }
88 else
89 {
90     return CopyFilledElements(array, filledElementsCount);
91 }
92 }
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
96 {
97     var finalArray = new TLink[filledElementsCount];
98     for (int i = 0, j = 0; i < array.Length; i++)
99     {
100         if (!_equalityComparer.Equals(array[i], default))
101         {
102             finalArray[j] = array[i];
103             j++;
104         }
105     }
106 #if USEARRAYPOOL
107     ArrayPool.Free(array);
108 #endif
109     return finalArray;
110 }
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 private static int CountFilledElements(TLink[] array)
114 {
115     var count = 0;
116     for (var i = 0; i < array.Length; i++)
117     {
118         if (!_equalityComparer.Equals(array[i], default))
119         {
120             count++;
121         }
122     }
123     return count;
124 }
125 }
126 }

```

1.88 ./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 }

```

```

24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     protected override IEnumerable<TLink> WalkContents(TLink element)
26     {
27         var parts = Links.GetLink(element);
28         for (var i = Links.Constants.IndexPart + 1; i < parts.Count; i++)
29         {
30             var part = parts[i];
31             if (IsElement(part))
32             {
33                 yield return part;
34             }
35         }
36     }
37 }
38 }

```

1.89 ./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)]
25         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
26             ↳ stack, links.IsPartialPoint) { }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public IEnumerable<TLink> Walk(TLink sequence)
30         {
31             _stack.Clear();
32             var element = sequence;
33             if (IsElement(element))
34             {
35                 yield return element;
36             }
37             else
38             {
39                 while (true)
40                 {
41                     if (IsElement(element))
42                     {
43                         if (_stack.IsEmpty)
44                         {
45                             break;
46                         }
47                         element = _stack.Pop();
48                         foreach (var output in WalkContents(element))
49                         {
50                             yield return output;
51                         }
52                         element = GetNextElementAfterPop(element);
53                     }
54                     else
55                     {
56                         _stack.Push(element);
57                         element = GetNextElementAfterPush(element);
58                     }
59                 }
60             }
61         }
62     }
63 }

```

```

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 ./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> : IStack<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly ILinks<TLink> _links;
15         private readonly TLink _stack;
16
17         public bool IsEmpty
18         {
19             [MethodImpl(MethodImplOptions.AggressiveInlining)]
20             get => _equalityComparer.Equals(Peek(), _stack);
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public Stack(ILinks<TLink> links, TLink stack)
25         {
26             _links = links;
27             _stack = stack;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         private TLink GetStackMarker() => _links.GetSource(_stack);
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         private TLink GetTop() => _links.GetTarget(_stack);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public TLink Peek() => _links.GetTarget(GetTop());
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public TLink Pop()
41         {
42             var element = Peek();
43             if (!_equalityComparer.Equals(element, _stack))
44             {
45                 var top = GetTop();
46                 var previousTop = _links.GetSource(top);
47                 _links.Update(_stack, GetStackMarker(), previousTop);
48                 _links.Delete(top);
49             }
50             return element;
51         }
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
55             ↪ _links.GetOrCreate(GetTop(), element));
56     }
57 }

```

1.91 ./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 }

```

1.92 ./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 }

```

```

63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
        ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
        ↳ Unsync.Update);
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     public void Delete(IList<TLinkAddress> restrictions) =>
        ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
68
69     //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
        ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
70     //{
71     //    if (restriction != null && substitution != null &&
        ↳ !substitution.EqualTo(restriction))
72     //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
        ↳ substitution, substitutedHandler, Unsync.Trigger);
73
74     //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
        ↳ substitutedHandler, Unsync.Trigger);
75     //}
76 }
77 }

```

1.93 ./Platform.Data.Doublets/UInt64LinksExtensions.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Singletons;
6  using Platform.Data.Doublets.Unicode;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets
11 {
12     public static class UInt64LinksExtensions
13     {
14         public static readonly LinksConstants<ulong> Constants =
            ↳ Default<LinksConstants<ulong>>.Instance;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
21         {
22             if (sequence == null)
23             {
24                 return false;
25             }
26             var constants = links.Constants;
27             for (var i = 0; i < sequence.Length; i++)
28             {
29                 if (sequence[i] == constants.Any)
30                 {
31                     return true;
32                 }
33             }
34             return false;
35         }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
            ↳ Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
            ↳ false)
39         {
40             var sb = new StringBuilder();
41             var visited = new HashSet<ulong>();
42             links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
                ↳ innerSb.Append(link.Index), renderIndex, renderDebug);
43             return sb.ToString();
44         }
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
            ↳ Func<Link<ulong>, bool> isElement, Action<StringBuilder, Link<ulong>> appendElement,
            ↳ bool renderIndex = false, bool renderDebug = false)
48         {
49             var sb = new StringBuilder();

```



```

50     var visited = new HashSet<ulong>();
51     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
    ↪ renderDebug);
52     return sb.ToString();
53 }
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
    ↪ HashSet<ulong> visited, ulong linkIndex, Func<Link<ulong>, bool> isElement,
    ↪ Action<StringBuilder, Link<ulong>> appendElement, bool renderIndex = false, bool
    ↪ renderDebug = false)
57 {
58     if (sb == null)
59     {
60         throw new ArgumentNullException(nameof(sb));
61     }
62     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
    ↪ Constants.Itself)
63     {
64         return;
65     }
66     if (links.Exists(linkIndex))
67     {
68         if (visited.Add(linkIndex))
69         {
70             sb.Append('(');
71             var link = new Link<ulong>(links.GetLink(linkIndex));
72             if (renderIndex)
73             {
74                 sb.Append(link.Index);
75                 sb.Append(':');
76             }
77             if (link.Source == link.Index)
78             {
79                 sb.Append(link.Index);
80             }
81             else
82             {
83                 var source = new Link<ulong>(links.GetLink(link.Source));
84                 if (isElement(source))
85                 {
86                     appendElement(sb, source);
87                 }
88                 else
89                 {
90                     links.AppendStructure(sb, visited, source.Index, isElement,
    ↪ appendElement, renderIndex);
91                 }
92             }
93             sb.Append(' ');
94             if (link.Target == link.Index)
95             {
96                 sb.Append(link.Index);
97             }
98             else
99             {
100                 var target = new Link<ulong>(links.GetLink(link.Target));
101                 if (isElement(target))
102                 {
103                     appendElement(sb, target);
104                 }
105                 else
106                 {
107                     links.AppendStructure(sb, visited, target.Index, isElement,
    ↪ appendElement, renderIndex);
108                 }
109             }
110             sb.Append(')');
111         }
112         else
113         {
114             if (renderDebug)
115             {
116                 sb.Append('*');
117             }
118             sb.Append(linkIndex);
119         }
120     }

```

```

121         else
122         {
123             if (renderDebug)
124             {
125                 sb.Append('~');
126             }
127             sb.Append(linkIndex);
128         }
129     }
130 }
131 }

```

1.94 ./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         ///
42         /// Или
43         ///
44         /// public struct TransitionHeader
45         /// {
46         ///     public ulong TransactionIdCombined;
47         ///     public ulong TimestampCombined;
48         ///
49         ///     public ulong TransactionId
50         ///     {
51         ///         get
52         ///         {
53         ///             return (ulong) mask & TransactionIdCombined;
54         ///         }
55         ///     }
56         ///
57         ///     public UniqueTimestamp Timestamp
58         ///     {
59         ///         get
60         ///         {
61         ///             return (UniqueTimestamp)mask & TransactionIdCombined;
62         ///         }
63         ///     }
64         ///
65         ///     public TransactionItemType Type
66         ///     {

```

```

67     ///         get
68     ///         {
69     ///             // Использовать по одному биту из TransactionId и Timestamp,
70     ///             // для значения в 2 бита, которое представляет тип операции
71     ///             throw new NotImplementedException();
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
96         ↪ transactionId, Link<ulong> before, Link<ulong> after)
97         {
98             TransactionId = transactionId;
99             Before = before;
100             After = after;
101             Timestamp = uniqueTimestampFactory.Create();
102         }
103
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
106         ↪ transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
107         ↪ before, default) { }
108
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
111         ↪ transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
112         ↪ }
113
114         [MethodImpl(MethodImplOptions.AggressiveInlining)]
115         public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
116         ↪ {After}";
117
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
119         public override bool Equals(object obj) => obj is Transition transition ?
120         ↪ Equals(transition) : false;
121
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         public override int GetHashCode() => (TransactionId, Before, After,
124         ↪ Timestamp).GetHashCode();
125
126         [MethodImpl(MethodImplOptions.AggressiveInlining)]
127         public bool Equals(Transition other) => TransactionId == other.TransactionId &&
128         ↪ Before == other.Before && After == other.After && Timestamp == other.Timestamp;
129
130         [MethodImpl(MethodImplOptions.AggressiveInlining)]
131         public static bool operator ==(Transition left, Transition right) =>
132         ↪ left.Equals(right);
133
134         [MethodImpl(MethodImplOptions.AggressiveInlining)]
135         public static bool operator !=(Transition left, Transition right) => !(left ==
136         ↪ right);
137     }
138
139     /// <remarks>
140     /// Другие варианты реализации транзакций (атомарности):
141     /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
142     ↪ Target)) и индексов.
143     /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
144     ↪ потребуется решить вопрос

```

```

132     /// со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
133     ↪ пересечениями идентификаторов.
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
144     ↪ константой.
145     /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
146     ↪ создавая задержку.
147     /// На жёстком диске:
148     /// Минусы:
149     /// 1. Длительный отклик, на запись каждой трансформации.
150     /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
151     /// -> Это может решаться упаковкой/исключением дублирующих операций.
152     /// -> Также это может решаться тем, что короткие транзакции вообще
153     /// не будут записываться в случае отката.
154     /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
155     ↪ операции (трансформации)
156     /// будут записаны в лог.
157     /// </remarks>
158     public class Transaction : DisposableBase
159     {
160         private readonly Queue<Transition> _transitions;
161         private readonly UInt64LinksTransactionsLayer _layer;
162         public bool IsCommitted { get; private set; }
163         public bool IsReverted { get; private set; }
164
165         [MethodImpl(MethodImplOptions.AggressiveInlining)]
166         public Transaction(UInt64LinksTransactionsLayer layer)
167         {
168             _layer = layer;
169             if (_layer._currentTransactionId != 0)
170             {
171                 throw new NotSupportedException("Nested transactions not supported.");
172             }
173             IsCommitted = false;
174             IsReverted = false;
175             _transitions = new Queue<Transition>();
176             SetCurrentTransaction(layer, this);
177         }
178
179         [MethodImpl(MethodImplOptions.AggressiveInlining)]
180         public void Commit()
181         {
182             EnsureTransactionAllowsWriteOperations(this);
183             while (_transitions.Count > 0)
184             {
185                 var transition = _transitions.Dequeue();
186                 _layer._transitions.Enqueue(transition);
187             }
188             _layer._lastCommittedTransactionId = _layer._currentTransactionId;
189             IsCommitted = true;
190         }
191
192         [MethodImpl(MethodImplOptions.AggressiveInlining)]
193         private void Revert()
194         {
195             EnsureTransactionAllowsWriteOperations(this);
196             var transitionsToRevert = new Transition[_transitions.Count];
197             _transitions.CopyTo(transitionsToRevert, 0);
198             for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
199             {
200                 _layer.RevertTransition(transitionsToRevert[i]);
201             }
202             IsReverted = true;
203         }
204
205         [MethodImpl(MethodImplOptions.AggressiveInlining)]
206         public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
207             ↪ Transaction transaction)

```

```

206     {
207         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
208         layer._currentTransactionTransitions = transaction._transitions;
209         layer._currentTransaction = transaction;
210     }
211
212     [MethodImpl(MethodImplOptions.AggressiveInlining)]
213     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
214     {
215         if (transaction.IsReverted)
216         {
217             throw new InvalidOperationException("Transation is reverted.");
218         }
219         if (transaction.IsCommitted)
220         {
221             throw new InvalidOperationException("Transation is committed.");
222         }
223     }
224
225     [MethodImpl(MethodImplOptions.AggressiveInlining)]
226     protected override void Dispose(bool manual, bool wasDisposed)
227     {
228         if (!wasDisposed && _layer != null && !_layer.IsDisposed)
229         {
230             if (!IsCommitted && !IsReverted)
231             {
232                 Revert();
233             }
234             _layer.ResetCurrentTransation();
235         }
236     }
237 }
238
239 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
240
241 private readonly string _logAddress;
242 private readonly FileStream _log;
243 private readonly Queue<Transition> _transitions;
244 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
245 private Task _transitionsPusher;
246 private Transition _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>();

```

```

283     _transitionsPusher = new Task(TransitionsPusher);
284     _transitionsPusher.Start();
285 }
286
287 [MethodImpl(MethodImplOptions.AggressiveInlining)]
288 public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
289
290 [MethodImpl(MethodImplOptions.AggressiveInlining)]
291 public override ulong Create(IList<ulong> restrictions)
292 {
293     var createdLinkIndex = Links.Create();
294     var createdLink = new Link<ulong>(Links.GetLink(createdLinkIndex));
295     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
296         ↪ default, createdLink));
297     return createdLinkIndex;
298 }
299
300 [MethodImpl(MethodImplOptions.AggressiveInlining)]
301 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
302 {
303     var linkIndex = restrictions[Constants.IndexPart];
304     var beforeLink = new Link<ulong>(Links.GetLink(linkIndex));
305     linkIndex = Links.Update(restrictions, substitution);
306     var afterLink = new Link<ulong>(Links.GetLink(linkIndex));
307     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
308         ↪ beforeLink, afterLink));
309     return linkIndex;
310 }
311
312 [MethodImpl(MethodImplOptions.AggressiveInlining)]
313 public override void Delete(IList<ulong> restrictions)
314 {
315     var link = restrictions[Constants.IndexPart];
316     var deletedLink = new Link<ulong>(Links.GetLink(link));
317     Links.Delete(link);
318     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
319         ↪ deletedLink, default));
320 }
321
322 [MethodImpl(MethodImplOptions.AggressiveInlining)]
323 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
324     ↪ _transitions;
325
326 [MethodImpl(MethodImplOptions.AggressiveInlining)]
327 private void CommitTransition(Transition transition)
328 {
329     if (_currentTransaction != null)
330     {
331         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
332     }
333     var transitions = GetCurrentTransitions();
334     transitions.Enqueue(transition);
335 }
336
337 [MethodImpl(MethodImplOptions.AggressiveInlining)]
338 private void RevertTransition(Transition transition)
339 {
340     if (transition.After.IsNull()) // Revert Deletion with Creation
341     {
342         Links.Create();
343     }
344     else if (transition.Before.IsNull()) // Revert Creation with Deletion
345     {
346         Links.Delete(transition.After.Index);
347     }
348     else // Revert Update
349     {
350         Links.Update(new[] { transition.After.Index, transition.Before.Source,
351             ↪ transition.Before.Target });
352     }
353 }
354
355 [MethodImpl(MethodImplOptions.AggressiveInlining)]
356 private void ResetCurrentTransation()
357 {
358     _currentTransactionId = 0;
359     _currentTransactionTransitions = null;
360     _currentTransaction = null;

```

```

356     }
357
358     [MethodImpl(MethodImplOptions.AggressiveInlining)]
359     private void PushTransitions()
360     {
361         if (_log == null || _transitions == null)
362         {
363             return;
364         }
365         for (var i = 0; i < _transitions.Count; i++)
366         {
367             var transition = _transitions.Dequeue();
368
369             _log.Write(transition);
370             _lastCommittedTransition = transition;
371         }
372     }
373
374     [MethodImpl(MethodImplOptions.AggressiveInlining)]
375     private void TransitionsPusher()
376     {
377         while (!IsDisposed && _transitionsPusher != null)
378         {
379             Thread.Sleep(DefaultPushDelay);
380             PushTransitions();
381         }
382     }
383
384     [MethodImpl(MethodImplOptions.AggressiveInlining)]
385     public Transaction BeginTransaction() => new Transaction(this);
386
387     [MethodImpl(MethodImplOptions.AggressiveInlining)]
388     private void DisposeTransitions()
389     {
390         try
391         {
392             var pusher = _transitionsPusher;
393             if (pusher != null)
394             {
395                 _transitionsPusher = null;
396                 pusher.Wait();
397             }
398             if (_transitions != null)
399             {
400                 PushTransitions();
401             }
402             _log.DisposeIfPossible();
403             FileHelpers.WriteFirst(_logAddress, _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

```

1.95 ./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Converters;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Unicode
7 {

```

```

8     public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<char, TLink>
9     {
10         private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
    ↪ UncheckedConverter<char, TLink>.Default;

11
12         private readonly IConverter<TLink> _addressToNumberConverter;
13         private readonly TLink _unicodeSymbolMarker;

14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
    ↪ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
17         {
18             _addressToNumberConverter = addressToNumberConverter;
19             _unicodeSymbolMarker = unicodeSymbolMarker;
20         }

21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public TLink Convert(char source)
24         {
25             var unaryNumber =
    ↪ _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
26             return Links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
27         }
28     }
29 }

```

1.96 ./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 < source.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 }

```

1.97 ./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
61             // 0 - null link
62             // 1 - nil character (0 character)
63             // ...
64             // 65536 (0(1) + 65535 = 65536 possible values)
65
66             [MethodImpl(MethodImplOptions.AggressiveInlining)]
67             public static ulong FromCharToLink(char character) => (ulong)character + 1;
68
69             [MethodImpl(MethodImplOptions.AggressiveInlining)]
70             public static char FromLinkToChar(ulong link) => (char)(link - 1);
71
72             [MethodImpl(MethodImplOptions.AggressiveInlining)]
73             public static bool IsCharLink(ulong link) => link <= MapSize;
74
75             [MethodImpl(MethodImplOptions.AggressiveInlining)]
76             public static string FromLinksToString(IList<ulong> linksList)
77             {
78                 var sb = new StringBuilder();
79                 for (int i = 0; i < linksList.Count; i++)
80                 {
81                     sb.Append(FromLinkToChar(linksList[i]));
82                 }
83                 return sb.ToString();
84             }
85
86             [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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
103     ↪ chars.Length);
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
106 {
107     // char array to ulong array
108     var linksSequence = new ulong[count];
109     for (var i = 0; i < count; i++)
110     {
111         linksSequence[i] = FromCharToLink(chars[i]);
112     }
113     return linksSequence;
114 }
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 public static ulong[] FromStringToLinkArray(string sequence)
117 {
118     // char array to ulong array
119     var linksSequence = new ulong[sequence.Length];
120     for (var i = 0; i < sequence.Length; i++)
121     {
122         linksSequence[i] = FromCharToLink(sequence[i]);
123     }
124     return linksSequence;
125 }
126 [MethodImpl(MethodImplOptions.AggressiveInlining)]
127 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
128 {
129     var result = new List<ulong[]>();
130     var offset = 0;
131     while (offset < sequence.Length)
132     {
133         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
134         var relativeLength = 1;
135         var absoluteLength = offset + relativeLength;
136         while (absoluteLength < sequence.Length &&
137             ↪ currentCategory ==
138             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
139         {
140             relativeLength++;
141             absoluteLength++;
142         }
143         // char array to ulong array
144         var innerSequence = new ulong[relativeLength];
145         var maxLength = offset + relativeLength;
146         for (var i = offset; i < maxLength; i++)
147         {
148             innerSequence[i - offset] = FromCharToLink(sequence[i]);
149         }
150         result.Add(innerSequence);
151         offset += relativeLength;
152     }
153     return result;
154 }
155 [MethodImpl(MethodImplOptions.AggressiveInlining)]
156 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
157 {
158     var result = new List<ulong[]>();
159     var offset = 0;

```

```

162     while (offset < array.Length)
163     {
164         var relativeLength = 1;
165         if (array[offset] <= LastCharLink)
166         {
167             var currentCategory =
168                 ↳ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
169             var absoluteLength = offset + relativeLength;
170             while (absoluteLength < array.Length &&
171                 array[absoluteLength] <= LastCharLink &&
172                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
173                     ↳ array[absoluteLength])))
174             {
175                 relativeLength++;
176                 absoluteLength++;
177             }
178         }
179         else
180         {
181             var absoluteLength = offset + relativeLength;
182             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
183             {
184                 relativeLength++;
185                 absoluteLength++;
186             }
187             // copy array
188             var innerSequence = new ulong[relativeLength];
189             var maxLength = offset + relativeLength;
190             for (var i = offset; i < maxLength; i++)
191             {
192                 innerSequence[i - offset] = array[i];
193             }
194             result.Add(innerSequence);
195             offset += relativeLength;
196         }
197     }
198     return result;
199 }

```

1.98 ./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 }

```

1.99 ./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>,
    ↳ IConverter<TLink, string>
13 {
14     private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
15     private readonly ISequenceWalker<TLink> _sequenceWalker;
16     private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;
17
18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
    ↳ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
    ↳ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
20     {
21         _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
22         _sequenceWalker = sequenceWalker;
23         _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
24     }
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     public string Convert(TLink source)
28     {
29         if(!_unicodeSequenceCriterionMatcher.IsMatched(source))
30         {
31             throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
    ↳ not a unicode sequence.");
32         }
33         var sequence = Links.GetSource(source);
34         var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
    ↳ Convert).ToArray();
35         return new string(charArray);
36     }
37 }
38 }

```

1.100 ./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>,
    ↳ ICriterionMatcher<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↳ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _unicodeSymbolMarker;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public UnicodeSymbolCriterionMatcher(ILinks<TLink> links, TLink unicodeSymbolMarker) :
    ↳ base(links) => _unicodeSymbolMarker = unicodeSymbolMarker;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public bool IsMatched(TLink link) => _equalityComparer.Equals(Links.GetTarget(link),
    ↳ _unicodeSymbolMarker);
20     }
21 }

```

1.101 ./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>,
    ↳ IConverter<TLink, char>
11     {
12         private static readonly UncheckedConverter<TLink, ushort> _addressToUInt16Converter =
    ↳ UncheckedConverter<TLink, ushort>.Default;
13
14         private readonly IConverter<TLink> _numberToAddressConverter;
15         private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

18     public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
    ↪     numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
    ↪     base(links)
19     {
20         _numberToAddressConverter = numberToAddressConverter;
21         _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
22     }
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     public char Convert(TLink source)
26     {
27         if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
28         {
29             throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
    ↪             not a unicode symbol.");
30         }
31         return (char)_addressToUInt16Converter.Convert(_numberToAddressConverter.Convert(Lin
    ↪         ks.GetSource(source)));
32     }
33 }
34 }

```

1.102 ./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<ulong>
11         {
12             public int Compare(ulong x, ulong y) => x.CompareTo(y);
13         }
14
15         private static int Compare(ulong x, ulong y) => x.CompareTo(y);
16
17         [Fact]
18         public static void GreaterOrEqualPerformanceTest()
19         {
20             const int N = 1000000;
21
22             ulong x = 10;
23             ulong y = 500;
24
25             bool result = false;
26
27             var ts1 = Performance.Measure(() =>
28             {
29                 for (int i = 0; i < N; i++)
30                 {
31                     result = Compare(x, y) >= 0;
32                 }
33             });
34
35             var comparer1 = Comparer<ulong>.Default;
36
37             var ts2 = Performance.Measure(() =>
38             {
39                 for (int i = 0; i < N; i++)
40                 {
41                     result = comparer1.Compare(x, y) >= 0;
42                 }
43             });
44
45             Func<ulong, ulong, int> compareReference = comparer1.Compare;
46
47             var ts3 = Performance.Measure(() =>
48             {
49                 for (int i = 0; i < N; i++)
50                 {
51                     result = compareReference(x, y) >= 0;
52                 }
53             });
54
55             var comparer2 = new UInt64Comparer();
56
57             var ts4 = Performance.Measure(() =>

```

```

58     {
59         for (int i = 0; i < N; i++)
60         {
61             result = comparer2.Compare(x, y) >= 0;
62         }
63     });
64
65     Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");
66 }
67 }
68 }

```

1.103 ./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             public int GetHashCode(ulong obj) => obj.GetHashCode();
15         }
16
17         private static bool Equals1<T>(T x, T y) => Equals(x, y);
18
19         private static bool Equals2<T>(T x, T y) => x.Equals(y);
20
21         private static bool Equals3(ulong x, ulong y) => x == y;
22
23         [Fact]
24         public static void EqualsPerfomanceTest()
25         {
26             const int N = 1000000;
27
28             ulong x = 10;
29             ulong y = 500;
30
31             bool result = false;
32
33             var ts1 = Performance.Measure(() =>
34             {
35                 for (int i = 0; i < N; i++)
36                 {
37                     result = Equals1(x, y);
38                 }
39             });
40
41             var ts2 = Performance.Measure(() =>
42             {
43                 for (int i = 0; i < N; i++)
44                 {
45                     result = Equals2(x, y);
46                 }
47             });
48
49             var ts3 = Performance.Measure(() =>
50             {
51                 for (int i = 0; i < N; i++)
52                 {
53                     result = Equals3(x, y);
54                 }
55             });
56
57             var equalityComparer1 = EqualityComparer<ulong>.Default;
58
59             var ts4 = Performance.Measure(() =>
60             {
61                 for (int i = 0; i < N; i++)
62                 {
63                     result = equalityComparer1.Equals(x, y);
64                 }
65             });
66
67             var equalityComparer2 = new UInt64EqualityComparer();

```

```

68
69     var ts5 = Performance.Measure(() =>
70     {
71         for (int i = 0; i < N; i++)
72         {
73             result = equalityComparer2.Equals(x, y);
74         }
75     });
76
77     Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;
78
79     var ts6 = Performance.Measure(() =>
80     {
81         for (int i = 0; i < N; i++)
82         {
83             result = equalityComparer3(x, y);
84         }
85     });
86
87     var comparer = Comparer<ulong>.Default;
88
89     var ts7 = Performance.Measure(() =>
90     {
91         for (int i = 0; i < N; i++)
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 }

```

1.104 ./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));

```

```

36         Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
        ↪ tMultipleRandomCreationsAndDeletions(100));
37     }
38
39     private static void Using<TLink>(Action<ILinks<TLink>> action)
40     {
41         using (var scope = new Scope<Types<HeapResizableDirectMemory,
        ↪ ResizableDirectMemoryLinks<TLink>>>())
42         {
43             action(scope.Use<ILinks<TLink>>());
44         }
45     }
46 }
47 }

```

1.105 ./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 }

```

1.106 ./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 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.
67 Pulvinar mattis nunc sed blandit libero volutpat.
68 Cras fermentum odio eu feugiat pretium nibh ipsum.
69 In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
70 Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
71 A iaculis at erat pellentesque.
72 Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
73 Eget lorem dolor sed viverra ipsum nunc.
74 Leo a diam sollicitudin tempor id eu.
75 Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";
76
77 [Fact]
78 public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
79 {
80     using (var scope = new TempLinksTestScope(useSequences: false))
81     {
82         var links = scope.Links;
83         var constants = links.Constants;
84
85         links.UseUnicode();
86
87         var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
88
89         var meaningRoot = links.CreatePoint();
90         var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
91         var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
92         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
93             ↳ constants.Itself);
94
95         var unaryNumberToAddressConverter = new
96             ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
97         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
98         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
99             ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
100         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
101             ↳ frequencyPropertyMarker, frequencyMarker);
102         var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
103             ↳ frequencyPropertyOperator, frequencyIncrementer);
104         var linkToItsFrequencyNumberConverter = new
105             ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
106             ↳ unaryNumberToAddressConverter);
107         var sequenceToItsLocalElementLevelsConverter = new
108             ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
109             ↳ linkToItsFrequencyNumberConverter);
110         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
111             ↳ sequenceToItsLocalElementLevelsConverter);
112
113         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
114             ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
115
116         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
117             ↳ index, optimalVariantConverter);
118     }
119 }
120 [Fact]

```

```

110 public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
111 {
112     using (var scope = new TempLinksTestScope(useSequences: false))
113     {
114         var links = scope.Links;
115
116         links.UseUnicode();
117
118         var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
119
120         var totalSequenceSymbolFrequencyCounter = new
121             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
122
123         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
124             ↪ totalSequenceSymbolFrequencyCounter);
125
126         var index = new
127             ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
128         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequency
129             ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
130
131         var sequenceToItsLocalElementLevelsConverter = new
132             ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
133             ↪ linkToItsFrequencyNumberConverter);
134         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
135             ↪ sequenceToItsLocalElementLevelsConverter);
136
137         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
138             ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
139
140         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
141             ↪ index, optimalVariantConverter);
142     }
143 }
144
145 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
146     ↪ SequenceToItsLocalElementLevelsConverter<ulong>
147     ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
148     ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
149 {
150     index.Add(sequence);
151
152     var optimalVariant = optimalVariantConverter.Convert(sequence);
153
154     var readSequence1 = sequences.ToList(optimalVariant);
155
156     Assert.True(sequence.SequenceEqual(readSequence1));
157 }
158
159 [Fact]
160 public static void SavedSequencesOptimizationTest()
161 {
162     LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
163         ↪ (long.MaxValue + 1UL, ulong.MaxValue));
164
165     using (var memory = new HeapResizableDirectMemory())
166     using (var disposableLinks = new UInt64ResizableDirectMemoryLinks(memory,
167         ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep, constants,
168         ↪ useAvlBasedIndex: false))
169     {
170         var links = new UInt64Links(disposableLinks);
171
172         var root = links.CreatePoint();
173
174         //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
175         var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
176
177         var unicodeSymbolMarker = links.GetOrCreate(root,
178             ↪ addressToNumberConverter.Convert(1));
179         var unicodeSequenceMarker = links.GetOrCreate(root,
180             ↪ addressToNumberConverter.Convert(2));
181
182         var totalSequenceSymbolFrequencyCounter = new
183             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
184         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
185             ↪ totalSequenceSymbolFrequencyCounter);
186         var index = new
187             ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);

```

```

168     var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
169     var sequenceToItsLocalElementLevelsConverter = new
170     ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
171     ↪ linkToItsFrequencyNumberConverter);
172     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
173     ↪ sequenceToItsLocalElementLevelsConverter);
174
175     var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
176     ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
177
178     var unicodeSequencesOptions = new SequencesOptions<ulong>()
179     {
180         UseSequenceMarker = true,
181         SequenceMarkerLink = unicodeSequenceMarker,
182         UseIndex = true,
183         Index = index,
184         LinksToSequenceConverter = optimalVariantConverter,
185         Walker = walker,
186         UseGarbageCollection = true
187     };
188
189     var unicodeSequences = new Sequences.Sequences(new
190     ↪ SynchronizedLinks<ulong>(links), unicodeSequencesOptions);
191
192     // Create some sequences
193     var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
194     ↪ StringSplitOptions.RemoveEmptyEntries);
195     var arrays = strings.Select(x => x.Select(y =>
196     ↪ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
197     for (int i = 0; i < arrays.Length; i++)
198     {
199         unicodeSequences.Create(arrays[i].ShiftRight());
200     }
201
202     var linksCountAfterCreation = links.Count();
203
204     // get list of sequences links
205     // for each sequence link
206     // create new sequence version
207     // if new sequence is not the same as sequence link
208     // delete sequence link
209     // collect garbage
210     unicodeSequences.CompactAll();
211
212     var linksCountAfterCompactification = links.Count();
213
214     Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
215 }
216 }
217 }

```

1.107 ./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<ulong> {
24                     ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
25
26                 var sequence = new ulong[sequenceLength];

```

```

26     for (var i = 0; i < sequenceLength; i++)
27     {
28         sequence[i] = links.Create();
29     }
30
31     var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
32
33     var sw1 = Stopwatch.StartNew();
34     var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
35
36     var sw2 = Stopwatch.StartNew();
37     var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();
38
39     var sw3 = Stopwatch.StartNew();
40     var readSequence2 = new List<ulong>();
41     SequenceWalker.WalkRight(balancedVariant,
42                             links.GetSource,
43                             links.GetTarget,
44                             links.IsPartialPoint,
45                             readSequence2.Add);
46
47     sw3.Stop();
48
49     Assert.True(sequence.SequenceEqual(readSequence1));
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 }

```

1.108 ./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                 {

```

```

36         var link = memoryAdapter.Create();
37         memoryAdapter.Delete(link);
38     }
39
40     [Fact]
41     public static void NonexistentReferencesHeapMemoryTest()
42     {
43         using (var memory = new
44             ↳ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
45         using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
46             ↳ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
47         {
48             memoryAdapter.TestNonexistentReferences();
49         }
50
51         private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
52         {
53             var link = memoryAdapter.Create();
54             memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
55             var resultLink = _constants.Null;
56             memoryAdapter.Each(foundLink =>
57             {
58                 resultLink = foundLink[_constants.IndexPart];
59                 return _constants.Break;
60             }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
61             Assert.True(resultLink == link);
62             Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
63             memoryAdapter.Delete(link);
64         }
65     }

```

1.109 ./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 }

```

1.110 ./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             //[Fact]
64             //public void CUDTest()
65             //{

```

```

66     //     var tempFilename = Path.GetTempFileName();
67
68     //     const long sequenceLength = 8;
69
70     //     const ulong itself = LinksConstants.Itself;
71
72     //     using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
73     ↪ DefaultLinksSizeStep))
74     //     using (var links = new Links(memoryAdapter))
75     //     {
76     //         var sequence = new ulong[sequenceLength];
77     //         for (var i = 0; i < sequenceLength; i++)
78     //             sequence[i] = links.Create(itself, itself);
79
80     //         SequencesOptions o = new SequencesOptions();
81
82     //         // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
83     //         o.
84
85     //         var sequences = new Sequences(links);
86
87     //         var sw1 = Stopwatch.StartNew();
88     //         var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
89
90     //         var sw2 = Stopwatch.StartNew();
91     //         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
92
93     //         Assert.True(results1.Count > results2.Length);
94     //         Assert.True(sw1.Elapsed > sw2.Elapsed);
95
96     //         for (var i = 0; i < sequenceLength; i++)
97     //             links.Delete(sequence[i]);
98     //     }
99
100    //     File.Delete(tempFilename);
101    // }
102
103    [Fact]
104    public static void AllVariantsSearchTest()
105    {
106        const long sequenceLength = 8;
107
108        using (var scope = new TempLinksTestScope(useSequences: true))
109        {
110            var links = scope.Links;
111            var sequences = scope.Sequences;
112
113            var sequence = new ulong[sequenceLength];
114            for (var i = 0; i < sequenceLength; i++)
115            {
116                sequence[i] = links.Create();
117            }
118
119            var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
120
121            //for (int i = 0; i < createResults.Length; i++)
122            //    sequences.Create(createResults[i]);
123
124            var sw0 = Stopwatch.StartNew();
125            var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
126
127            var sw1 = Stopwatch.StartNew();
128            var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
129
130            var sw2 = Stopwatch.StartNew();
131            var searchResults2 = sequences.Each1(sequence); sw2.Stop();
132
133            var sw3 = Stopwatch.StartNew();
134            var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
135
136            var intersection0 = createResults.Intersect(searchResults0).ToList();
137            Assert.True(intersection0.Count == searchResults0.Count);
138            Assert.True(intersection0.Count == createResults.Length);
139
140            var intersection1 = createResults.Intersect(searchResults1).ToList();
141            Assert.True(intersection1.Count == searchResults1.Count);
142            Assert.True(intersection1.Count == createResults.Length);
143
144            var intersection2 = createResults.Intersect(searchResults2).ToList();
145            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 [Fact]
158 public static void BalancedVariantSearchTest()
159 {
160     const long sequenceLength = 200;
161
162     using (var scope = new TempLinksTestScope(useSequences: true))
163     {
164         var links = scope.Links;
165         var sequences = scope.Sequences;
166
167         var sequence = new ulong[sequenceLength];
168         for (var i = 0; i < sequenceLength; i++)
169         {
170             sequence[i] = links.Create();
171         }
172
173         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
174
175         var sw1 = Stopwatch.StartNew();
176         var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
177
178         var sw2 = Stopwatch.StartNew();
179         var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
180
181         var sw3 = Stopwatch.StartNew();
182         var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
183
184         // На количестве в 200 элементов это будет занимать вечность
185         //var sw4 = Stopwatch.StartNew();
186         //var searchResults4 = sequences.Each(sequence); sw4.Stop();
187
188         Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
189
190         Assert.True(searchResults3.Count == 1 && balancedVariant ==
191             ↪ searchResults3.First());
192
193         //Assert.True(sw1.Elapsed < sw2.Elapsed);
194
195         for (var i = 0; i < sequenceLength; i++)
196         {
197             links.Delete(sequence[i]);
198         }
199     }
200 }
201
202 [Fact]
203 public static void AllPartialVariantsSearchTest()
204 {
205     const long sequenceLength = 8;
206
207     using (var scope = new TempLinksTestScope(useSequences: true))
208     {
209         var links = scope.Links;
210         var sequences = scope.Sequences;
211
212         var sequence = new ulong[sequenceLength];
213         for (var i = 0; i < sequenceLength; i++)
214         {
215             sequence[i] = links.Create();
216         }
217
218         var createResults = sequences.CreateAllVariants2(sequence);
219
220         //var createResultsStrings = createResults.Select(x => x + ": " +
221             ↪ sequences.FormatSequence(x)).ToList();
222         //Global.Trash = createResultsStrings;

```



```

223     var partialSequence = new ulong[sequenceLength - 2];
224
225     Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
226
227     var sw1 = Stopwatch.StartNew();
228     var searchResults1 =
229         ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
230
231     var sw2 = Stopwatch.StartNew();
232     var searchResults2 =
233         ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
234
235     //var sw3 = Stopwatch.StartNew();
236     //var searchResults3 =
237         ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
238
239     var sw4 = Stopwatch.StartNew();
240     var searchResults4 =
241         ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
242
243     //Global.Trash = searchResults3;
244
245     //var searchResults1Strings = searchResults1.Select(x => x + ": " +
246         ↪ sequences.FormatSequence(x)).ToList();
247     //Global.Trash = searchResults1Strings;
248
249     var intersection1 = createResults.Intersect(searchResults1).ToList();
250     Assert.True(intersection1.Count == createResults.Length);
251
252     var intersection2 = createResults.Intersect(searchResults2).ToList();
253     Assert.True(intersection2.Count == createResults.Length);
254
255     var intersection4 = createResults.Intersect(searchResults4).ToList();
256     Assert.True(intersection4.Count == createResults.Length);
257
258     for (var i = 0; i < sequenceLength; i++)
259     {
260         links.Delete(sequence[i]);
261     }
262 }
263
264 [Fact]
265 public static void BalancedPartialVariantsSearchTest()
266 {
267     const long sequenceLength = 200;
268
269     using (var scope = new TempLinksTestScope(useSequences: true))
270     {
271         var links = scope.Links;
272         var sequences = scope.Sequences;
273
274         var sequence = new ulong[sequenceLength];
275         for (var i = 0; i < sequenceLength; i++)
276         {
277             sequence[i] = links.Create();
278         }
279
280         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
281
282         var balancedVariant = balancedVariantConverter.Convert(sequence);
283
284         var partialSequence = new ulong[sequenceLength - 2];
285
286         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
287
288         var sw1 = Stopwatch.StartNew();
289         var searchResults1 =
290             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
291
292         var sw2 = Stopwatch.StartNew();
293         var searchResults2 =
294             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
295
296         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
297
298         Assert.True(searchResults2.Count == 1 && balancedVariant ==
299             ↪ searchResults2.First());
300
301         for (var i = 0; i < sequenceLength; i++)

```

```

295         {
296             links.Delete(sequence[i]);
297         }
298     }
299 }
300
301 [Fact(Skip = "Correct implementation is pending")]
302 public static void PatternMatchTest()
303 {
304     var zeroOrMany = Sequences.Sequences.ZeroOrMany;
305
306     using (var scope = new TempLinksTestScope(useSequences: true))
307     {
308         var links = scope.Links;
309         var sequences = scope.Sequences;
310
311         var e1 = links.Create();
312         var e2 = links.Create();
313
314         var sequence = new[]
315         {
316             e1, e2, e1, e2 // mama / papa
317         };
318
319         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
320
321         var balancedVariant = balancedVariantConverter.Convert(sequence);
322
323         // 1: [1]
324         // 2: [2]
325         // 3: [1,2]
326         // 4: [1,2,1,2]
327
328         var doublet = links.GetSource(balancedVariant);
329
330         var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
331
332         Assert.True(matchedSequences1.Count == 0);
333
334         var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
335
336         Assert.True(matchedSequences2.Count == 0);
337
338         var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
339
340         Assert.True(matchedSequences3.Count == 0);
341
342         var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
343
344         Assert.Contains(doublet, matchedSequences4);
345         Assert.Contains(balancedVariant, matchedSequences4);
346
347         for (var i = 0; i < sequence.Length; i++)
348         {
349             links.Delete(sequence[i]);
350         }
351     }
352 }
353
354 [Fact]
355 public static void IndexTest()
356 {
357     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
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);

```

```

374         Assert.True(index.MightContain(sequence));
375     }
376 }
377 }
378
379 /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
    ↳ DO%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
    ↳ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
380 private static readonly string _exampleText =
381     @"([english
    ↳ version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
382
383 Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов
    ↳ (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там
    ↳ где есть место для нового начала? Разве пустота это не характеристика пространства?
    ↳ Пространство это то, что можно чем-то наполнить?
384
385 ![чёрное пространство, белое
    ↳ пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
    ↳ "чёрное пространство, белое пространство")](https://raw.githubusercontent.com/Konard/Links
    ↳ Platform/master/doc/Intro/1.png)
386
387 Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая
    ↳ форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?
388
389 ![чёрное пространство, чёрная
    ↳ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png
    ↳ "чёрное пространство, чёрная
    ↳ точка")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)
390
391 А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
    ↳ так? Инверсия? Отражение? Сумма?
392
393 ![белая точка, чёрная
    ↳ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png "белая
    ↳ точка, чёрная
    ↳ точка")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)
394
395 А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет
    ↳ если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?
    ↳ Грань? 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-animat
    ↳ ion-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
457         ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
458     var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
459         ↳ totalSequenceSymbolFrequencyCounter);
460     var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
461         ↳ balancedVariantConverter, doubletFrequenciesCache);
462
463     var compressedVariant = compressingConverter.Convert(sequence);
464
465     // 1: [1]      (1->1) point
466     // 2: [2]      (2->2) point
467     // 3: [1,2]    (1->2) doublet
468     // 4: [1,2,1,2] (3->3) doublet
469
470     Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
471     Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
472     Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
473     Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
474
475     var source = _constants.SourcePart;
476     var target = _constants.TargetPart;
477
478     Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
479     Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
480     Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
481     Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
482
483     // 4 - length of sequence
484     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
485         ↳ == sequence[0]);
486     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
487         ↳ == sequence[1]);
488     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
489         ↳ == sequence[2]);
490     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
491         ↳ == sequence[3]);
492 }
493
494 [Fact]
495 public static void CompressionEfficiencyTest()
496 {
497     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
498         ↳ StringSplitOptions.RemoveEmptyEntries);
499     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
500     var totalCharacters = arrays.Select(x => x.Length).Sum();
501
502     using (var scope1 = new TempLinksTestScope(useSequences: true))
503     using (var scope2 = new TempLinksTestScope(useSequences: true))
504     using (var scope3 = new TempLinksTestScope(useSequences: true))
505     {
506         scope1.Links.Unsync.UseUnicode();
507         scope2.Links.Unsync.UseUnicode();
508         scope3.Links.Unsync.UseUnicode();
509
510         var balancedVariantConverter1 = new
511             ↳ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
512         var totalSequenceSymbolFrequencyCounter = new
513             ↳ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
514         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
515             ↳ totalSequenceSymbolFrequencyCounter);
516         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
517             ↳ balancedVariantConverter1, linkFrequenciesCache1,
518             ↳ doInitialFrequenciesIncrement: false);
519
520         //var compressor2 = scope2.Sequences;
521         var compressor3 = scope3.Sequences;
522
523         var constants = Default<LinksConstants<ulong>>.Instance;
524
525         var sequences = compressor3;
526         //var meaningRoot = links.CreatePoint();
527         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
528         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);

```

```

517 //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
518     ↳ constants.Itself);
519
520 //var unaryNumberToAddressConverter = new
521     ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
522 //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
523     ↳ unaryOne);
524 //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
525     ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
526 //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
527     ↳ frequencyPropertyMarker, frequencyMarker);
528 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
529     ↳ frequencyPropertyOperator, frequencyIncrementer);
530 //var linkToItsFrequencyNumberConverter = new
531     ↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
532     ↳ unaryNumberToAddressConverter);
533
534 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
535     ↳ totalSequenceSymbolFrequencyCounter);
536
537 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque_
538     ↳ ncyNumberConverter<ulong>(linkFrequenciesCache3);
539
540 var sequenceToItsLocalElementLevelsConverter = new
541     ↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
542     ↳ linkToItsFrequencyNumberConverter);
543 var optimalVariantConverter = new
544     ↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
545     ↳ sequenceToItsLocalElementLevelsConverter);
546
547 var compressed1 = new ulong[arrays.Length];
548 var compressed2 = new ulong[arrays.Length];
549 var compressed3 = new ulong[arrays.Length];
550
551 var START = 0;
552 var END = arrays.Length;
553
554 //for (int i = START; i < END; i++)
555 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
556
557 var initialCount1 = scope2.Links.Unsync.Count();
558
559 var sw1 = Stopwatch.StartNew();
560
561 for (int i = START; i < END; i++)
562 {
563     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
564     compressed1[i] = compressor1.Convert(arrays[i]);
565 }
566
567 var elapsed1 = sw1.Elapsed;
568
569 var balancedVariantConverter2 = new
570     ↳ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
571
572 var initialCount2 = scope2.Links.Unsync.Count();
573
574 var sw2 = Stopwatch.StartNew();
575
576 for (int i = START; i < END; i++)
577 {
578     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
579 }
580
581 var elapsed2 = sw2.Elapsed;
582
583 for (int i = START; i < END; i++)
584 {
585     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
586 }
587
588 var initialCount3 = scope3.Links.Unsync.Count();
589
590 var sw3 = Stopwatch.StartNew();
591
592 for (int i = START; i < END; i++)
593 {
594     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
595     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);

```

```

581 }
582
583 var elapsed3 = sw3.Elapsed;
584
585 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
    ↳ Optimal variant: {elapsed3}");
586
587 // Assert.True(elapsed1 > elapsed2);
588
589 // Checks
590 for (int i = START; i < END; i++)
591 {
592     var sequence1 = compressed1[i];
593     var sequence2 = compressed2[i];
594     var sequence3 = compressed3[i];
595
596     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
    ↳ scope1.Links.Unsync);
597
598     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
    ↳ scope2.Links.Unsync);
599
600     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
    ↳ scope3.Links.Unsync);
601
602     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
    ↳ link.IsPartialPoint());
603     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
    ↳ link.IsPartialPoint());
604     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
    ↳ link.IsPartialPoint());
605
606     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
    ↳ arrays[i].Length > 3)
607     //    Assert.False(structure1 == structure2);
608     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
    ↳ arrays[i].Length > 3)
609     //    Assert.False(structure3 == structure2);
610
611     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
612     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
613 }
614
615 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
    ↳ totalCharacters);
616 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
    ↳ totalCharacters);
617 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
    ↳ totalCharacters);
618
619 Console.WriteLine($"{{(double)(scope1.Links.Unsync.Count() - initialCount1) /
    ↳ totalCharacters}} | {{(double)(scope2.Links.Unsync.Count() - initialCount2) /
    ↳ totalCharacters}} | {{(double)(scope3.Links.Unsync.Count() - initialCount3) /
    ↳ totalCharacters}}");
620
621 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
    ↳ scope2.Links.Unsync.Count() - initialCount2);
622 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
    ↳ scope2.Links.Unsync.Count() - initialCount2);
623
624 var duplicateProvider1 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
625 var duplicateProvider2 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
626 var duplicateProvider3 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
627
628 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
629 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
630 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
631
632 var duplicates1 = duplicateCounter1.Count();
633
634 ConsoleHelpers.Debug("-----");
635
636 var duplicates2 = duplicateCounter2.Count();
637
638 ConsoleHelpers.Debug("-----");

```

```

639         var duplicates3 = duplicateCounter3.Count();
640
641         Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
642
643         linkFrequenciesCache1.ValidateFrequencies();
644         linkFrequenciesCache3.ValidateFrequencies();
645     }
646 }
647
648 [Fact]
649 public static void CompressionStabilityTest()
650 {
651     // TODO: Fix bug (do a separate test)
652     //const ulong minNumbers = 0;
653     //const ulong maxNumbers = 1000;
654
655     const ulong minNumbers = 10000;
656     const ulong maxNumbers = 12500;
657
658     var strings = new List<string>();
659
660     for (ulong i = minNumbers; i < maxNumbers; i++)
661     {
662         strings.Add(i.ToString());
663     }
664
665     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
666     var totalCharacters = arrays.Select(x => x.Length).Sum();
667
668     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
669         ↪ SequencesOptions<ulong> { UseCompression = true,
670         ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
671     using (var scope2 = new TempLinksTestScope(useSequences: true))
672     {
673         scope1.Links.UseUnicode();
674         scope2.Links.UseUnicode();
675
676         //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
677         var compressor1 = scope1.Sequences;
678         var compressor2 = scope2.Sequences;
679
680         var compressed1 = new ulong[arrays.Length];
681         var compressed2 = new ulong[arrays.Length];
682
683         var sw1 = Stopwatch.StartNew();
684
685         var START = 0;
686         var END = arrays.Length;
687
688         // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
689         // Stability issue starts at 10001 or 11000
690         //for (int i = START; i < END; i++)
691         //{
692         //    var first = compressor1.Compress(arrays[i]);
693         //    var second = compressor1.Compress(arrays[i]);
694
695         //    if (first == second)
696         //        compressed1[i] = first;
697         //    else
698         //    {
699         //        // TODO: Find a solution for this case
700         //    }
701         //}
702
703         for (int i = START; i < END; i++)
704         {
705             var first = compressor1.Create(arrays[i].ShiftRight());
706             var second = compressor1.Create(arrays[i].ShiftRight());
707
708             if (first == second)
709             {
710                 compressed1[i] = first;
711             }
712             else
713             {
714                 // TODO: Find a solution for this case
715             }
716         }
717     }
718 }

```



```

716     var elapsed1 = sw1.Elapsed;
717
718     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
719
720     var sw2 = Stopwatch.StartNew();
721
722     for (int i = START; i < END; i++)
723     {
724         var first = balancedVariantConverter.Convert(arrays[i]);
725         var second = balancedVariantConverter.Convert(arrays[i]);
726
727         if (first == second)
728         {
729             compressed2[i] = first;
730         }
731     }
732
733     var elapsed2 = sw2.Elapsed;
734
735     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
736     ↪ {elapsed2}");
737
738     Assert.True(elapsed1 > elapsed2);
739
740     // Checks
741     for (int i = START; i < END; i++)
742     {
743         var sequence1 = compressed1[i];
744         var sequence2 = compressed2[i];
745
746         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
747         {
748             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
749             ↪ scope1.Links);
750
751             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
752             ↪ scope2.Links);
753
754             //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
755             ↪ link.IsPartialPoint());
756             //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
757             ↪ link.IsPartialPoint());
758
759             //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
760             ↪ arrays[i].Length > 3)
761             //    Assert.False(structure1 == structure2);
762
763             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
764         }
765     }
766
767     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
768     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
769
770     Debug.WriteLine($"{{(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 [Fact]
780 public static void RandomNumbersCompressionQualityTest()
781 {
782     const ulong N = 500;
783
784     //const ulong minNumbers = 10000;
785     //const ulong maxNumbers = 20000;
786
787     //var strings = new List<string>();
788
789     //for (ulong i = 0; i < N; i++)
790     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
791     ↪ maxNumbers).ToString());

```

```

786 var strings = new List<string>();
787
788 for (ulong i = 0; i < N; i++)
789 {
790     strings.Add(RandomHelpers.Default.NextUInt64().ToString());
791 }
792
793 strings = strings.Distinct().ToList();
794
795 var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
796 var totalCharacters = arrays.Select(x => x.Length).Sum();
797
798 using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
    ↳ SequencesOptions<ulong> { UseCompression = true,
    ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
    using (var scope2 = new TempLinksTestScope(useSequences: true))
    {
799         scope1.Links.UseUnicode();
800         scope2.Links.UseUnicode();
801
802         var compressor1 = scope1.Sequences;
803         var compressor2 = scope2.Sequences;
804
805         var compressed1 = new ulong[arrays.Length];
806         var compressed2 = new ulong[arrays.Length];
807
808         var sw1 = Stopwatch.StartNew();
809
810         var START = 0;
811         var END = arrays.Length;
812
813         for (int i = START; i < END; i++)
814         {
815             compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
816         }
817
818         var elapsed1 = sw1.Elapsed;
819
820         var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
821
822         var sw2 = Stopwatch.StartNew();
823
824         for (int i = START; i < END; i++)
825         {
826             compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
827         }
828
829         var elapsed2 = sw2.Elapsed;
830
831         Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
832             ↳ {elapsed2}");
833
834         Assert.True(elapsed1 > elapsed2);
835
836         // Checks
837         for (int i = START; i < END; i++)
838         {
839             var sequence1 = compressed1[i];
840             var sequence2 = compressed2[i];
841
842             if (sequence1 != _constants.Null && sequence2 != _constants.Null)
843             {
844                 var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
845                     ↳ scope1.Links);
846
847                 var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
848                     ↳ scope2.Links);
849
850                 Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
851             }
852         }
853
854         Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
855         Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
856
857         Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
    ↳ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
    ↳ totalCharacters}}");

```

```

858         // Can be worse than balanced variant
859         //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
860
861         //compressor1.ValidateFrequencies();
862     }
863 }
864
865 [Fact]
866 public static void AllTreeBreakDownAtSequencesCreationBugTest()
867 {
868     // Made out of AllPossibleConnectionsTest test.
869
870     //const long sequenceLength = 5; //100% bug
871     const long sequenceLength = 4; //100% bug
872     //const long sequenceLength = 3; //100% _no_bug (ok)
873
874     using (var scope = new TempLinksTestScope(useSequences: true))
875     {
876         var links = scope.Links;
877         var sequences = scope.Sequences;
878
879         var sequence = new ulong[sequenceLength];
880         for (var i = 0; i < sequenceLength; i++)
881         {
882             sequence[i] = links.Create();
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
896 [Fact]
897 public static void AllPossibleConnectionsTest()
898 {
899     const long sequenceLength = 5;
900
901     using (var scope = new TempLinksTestScope(useSequences: true))
902     {
903         var links = scope.Links;
904         var sequences = scope.Sequences;
905
906         var sequence = new ulong[sequenceLength];
907         for (var i = 0; i < sequenceLength; i++)
908         {
909             sequence[i] = links.Create();
910         }
911
912         var createResults = sequences.CreateAllVariants2(sequence);
913         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
914
915         for (var i = 0; i < 1; i++)
916         {
917             var sw1 = Stopwatch.StartNew();
918             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
919
920             var sw2 = Stopwatch.StartNew();
921             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
922
923             var sw3 = Stopwatch.StartNew();
924             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
925
926             var sw4 = Stopwatch.StartNew();
927             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
928
929             Global.Trash = searchResults3;
930             Global.Trash = searchResults4; //-V3008
931
932             var intersection1 = createResults.Intersect(searchResults1).ToList();
933             Assert.True(intersection1.Count == createResults.Length);
934
935             var intersection2 = reverseResults.Intersect(searchResults1).ToList();
936             Assert.True(intersection2.Count == reverseResults.Length);
937

```

```

938         var intersection0 = searchResults1.Intersect(searchResults2).ToList();
939         Assert.True(intersection0.Count == searchResults2.Count);
940
941         var intersection3 = searchResults2.Intersect(searchResults3).ToList();
942         Assert.True(intersection3.Count == searchResults3.Count);
943
944         var intersection4 = searchResults3.Intersect(searchResults4).ToList();
945         Assert.True(intersection4.Count == searchResults4.Count);
946     }
947
948     for (var i = 0; i < sequenceLength; i++)
949     {
950         links.Delete(sequence[i]);
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         var sequences = scope.Sequences;
963
964         var sequence = new ulong[sequenceLength];
965         for (var i = 0; i < sequenceLength; i++)
966         {
967             sequence[i] = links.Create();
968         }
969
970         var createResults = sequences.CreateAllVariants2(sequence);
971
972         //var reverseResults =
973         ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
974
975         for (var i = 0; i < 1; i++)
976         {
977             var linksTotalUsages1 = new ulong[links.Count() + 1];
978
979             sequences.CalculateAllUsages(linksTotalUsages1);
980
981             var linksTotalUsages2 = new ulong[links.Count() + 1];
982
983             sequences.CalculateAllUsages2(linksTotalUsages2);
984
985             var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
986             Assert.True(intersection1.Count == linksTotalUsages2.Length);
987         }
988
989         for (var i = 0; i < sequenceLength; i++)
990         {
991             links.Delete(sequence[i]);
992         }
993     }
994 }
995 }
996 }

```

1.111 ./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
    ↪ useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
    ↪ useLog) { }
19
20     public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
    ↪ true, bool useSequences = false, bool useLog = false)
21     {
22         _deleteFiles = deleteFiles;
23         TempFilename = Path.GetTempFileName();
24         TempTransactionLogFilename = Path.GetTempFileName();
25         var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
26         MemoryAdapter = useLog ? (ILinks<ulong>)new
    ↪ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
    ↪ coreMemoryAdapter;
27         Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
28         if (useSequences)
29         {
30             Sequences = new Sequences.Sequences(Links, sequencesOptions);
31         }
32     }
33
34     protected override void Dispose(bool manual, bool wasDisposed)
35     {
36         if (!wasDisposed)
37         {
38             Links.Unsync.DisposeIfPossible();
39             if (_deleteFiles)
40             {
41                 DeleteFiles();
42             }
43         }
44     }
45
46     public void DeleteFiles()
47     {
48         File.Delete(TempFilename);
49         File.Delete(TempTransactionLogFilename);
50     }
51 }
52 }

```

1.112 ./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 = 0UL;
166         var deleted = 0UL;
167         for (var i = 0; i < N; i++)
168         {
169             var linksCount = addressToUInt64Converter.Convert(links.Count());
170             var createPoint = random.NextBoolean();
171             if (linksCount > 2 && createPoint)
172             {
173                 var linksAddressRange = new Range<ulong>(1, linksCount);
174                 TLink source = uint64ToAddressConverter.Convert(random.NextUInt64(linksA_
    ↪ ddressRange));
175                 TLink target = uint64ToAddressConverter.Convert(random.NextUInt64(linksA_
    ↪ ddressRange));
176                 ↪ //-V3086
177                 var resultLink = links.GetOrCreate(source, target);
178                 if (comparer.Compare(resultLink,
    ↪ uint64ToAddressConverter.Convert(linksCount)) > 0)
179                 {
180                     created++;
181                 }
182             }
183             else
184             {
185                 links.Create();
186                 created++;
187             }
188         }
189         Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
190         for (var i = 0; i < N; i++)
191         {
192             TLink link = uint64ToAddressConverter.Convert((ulong)i + 1UL);
193             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 }

```

1.113 ./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 =
28             ↪ Default<LinksConstants<ulong>>.Instance;
29
30         private const long Iterations = 10 * 1024;
31
32         #region Concept
33
34         [Fact]
35         public static void MultipleCreateAndDeleteTest()
36         {
37             using (var scope = new Scope<Types<HeapResizableDirectMemory,
38                 ↪ UInt64ResizableDirectMemoryLinks>>())
39             {
40                 new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeletions(100);
41             }
42         }
43
44         [Fact]
45         public static void CascadeUpdateTest()
46         {
47             var itself = _constants.Itself;
48             using (var scope = new TempLinksTestScope(useLog: true))
49             {
50                 var links = scope.Links;
51
52                 var l1 = links.Create();
53                 var l2 = links.Create();
54
55                 l2 = links.Update(l2, l2, l1, l2);
56
57                 links.CreateAndUpdate(l2, itself);
58                 links.CreateAndUpdate(l2, itself);
59
60                 l2 = links.Update(l2, l1);
61
62                 links.Delete(l2);
63
64                 Global.Trash = links.Count();
65
66                 links.Unsync.DisposeIfPossible(); // Close links to access log

```



```

66         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
        ↪ 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>(scope
        ↪ e.TempTransactionLogFilename);
86     }
87 }
88
89 [Fact]
90 public static void TransactionAutoRevertedTest()
91 {
92     // Auto Reverted (Because no commit at transaction)
93     using (var scope = new TempLinksTestScope(useLog: true))
94     {
95         var links = scope.Links;
96         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
97         using (var transaction = transactionsLayer.BeginTransaction())
98         {
99             var l1 = links.Create();
100             var l2 = links.Create();
101
102             links.Update(l2, l2, l1, l2);
103         }
104
105         Assert.Equal(0UL, links.Count());
106
107         links.Unsync.DisposeIfPossible();
108
109         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
        ↪ cope.TempTransactionLogFilename);
110         Assert.Single(transitions);
111     }
112 }
113
114 [Fact]
115 public static void TransactionUserCodeErrorNoDataSavedTest()
116 {
117     // User Code Error (Autoreverted), no data saved
118     var itself = _constants.Itself;
119
120     TempLinksTestScope lastScope = null;
121     try
122     {
123         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
        ↪ useLog: true))
124         {
125             var links = scope.Links;
126             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
        ↪ atorBase<ulong>)links.Unsync).Links;
127             using (var transaction = transactionsLayer.BeginTransaction())
128             {
129                 var l1 = links.CreateAndUpdate(itself, itself);
130                 var l2 = links.CreateAndUpdate(itself, itself);
131
132                 l2 = links.Update(l2, l2, l1, l2);
133
134                 links.CreateAndUpdate(l2, itself);
135                 links.CreateAndUpdate(l2, itself);
136
137                 //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
        ↪ tion>(scope.TempTransactionLogFilename);
138

```

```

139         l2 = links.Update(l2, l1);
140
141         links.Delete(l2);
142
143         ExceptionThrower();
144
145         transaction.Commit();
146     }
147
148     Global.Trash = links.Count();
149 }
150
151 catch
152 {
153     Assert.False(lastScope == null);
154
155     var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
        ↳ astScope.TempTransactionLogFilename);
156
157     Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
        ↳ transitions[0].After.IsNull());
158
159     lastScope.DeleteFiles();
160 }
161
162 }
163
164 [Fact]
165 public static void TransactionUserCodeErrorSomeDataSavedTest()
166 {
167     // User Code Error (Autoreverted), some data saved
168     var itself = _constants.Itself;
169
170     TempLinksTestScope lastScope = null;
171     try
172     {
173         ulong l1;
174         ulong l2;
175
176         using (var scope = new TempLinksTestScope(useLog: true))
177         {
178             var links = scope.Links;
179             l1 = links.CreateAndUpdate(itself, itself);
180             l2 = links.CreateAndUpdate(itself, itself);
181
182             l2 = links.Update(l2, l2, l1, l2);
183
184             links.CreateAndUpdate(l2, itself);
185             links.CreateAndUpdate(l2, itself);
186
187             links.Unsync.DisposeIfPossible();
188
189             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
                ↳ scope.TempTransactionLogFilename);
190         }
191
192         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
            ↳ useLog: true))
193         {
194             var links = scope.Links;
195             var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
196             using (var transaction = transactionsLayer.BeginTransaction())
197             {
198                 l2 = links.Update(l2, l1);
199
200                 links.Delete(l2);
201
202                 ExceptionThrower();
203
204                 transaction.Commit();
205             }
206
207             Global.Trash = links.Count();
208         }
209     }
210     catch
211     {
212         Assert.False(lastScope == null);
213
214         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
            ↳ Scope.TempTransactionLogFilename);

```

```

214         lastScope.DeleteFiles();
215     }
216 }
217
218 [Fact]
219 public static void TransactionCommit()
220 {
221     var itself = _constants.Itself;
222
223     var tempDatabaseFilename = Path.GetTempFileName();
224     var tempTransactionLogFilename = Path.GetTempFileName();
225
226     // Commit
227     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
228         ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
229         ↪ tempTransactionLogFilename))
230     using (var links = new UInt64Links(memoryAdapter))
231     {
232         using (var transaction = memoryAdapter.BeginTransaction())
233         {
234             var l1 = links.CreateAndUpdate(itself, itself);
235             var l2 = links.CreateAndUpdate(itself, itself);
236
237             Global.Trash = links.Update(l2, l2, l1, l2);
238
239             links.Delete(l1);
240
241             transaction.Commit();
242         }
243
244         Global.Trash = links.Count();
245     }
246
247     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
248         ↪ sactionLogFilename);
249 }
250
251 [Fact]
252 public static void TransactionDamage()
253 {
254     var itself = _constants.Itself;
255
256     var tempDatabaseFilename = Path.GetTempFileName();
257     var tempTransactionLogFilename = Path.GetTempFileName();
258
259     // Commit
260     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
261         ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
262         ↪ tempTransactionLogFilename))
263     using (var links = new UInt64Links(memoryAdapter))
264     {
265         using (var transaction = memoryAdapter.BeginTransaction())
266         {
267             var l1 = links.CreateAndUpdate(itself, itself);
268             var l2 = links.CreateAndUpdate(itself, itself);
269
270             Global.Trash = links.Update(l2, l2, l1, l2);
271
272             links.Delete(l1);
273
274             transaction.Commit();
275         }
276
277         Global.Trash = links.Count();
278     }
279
280     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
281         ↪ sactionLogFilename);
282
283     // Damage database
284     FileHelpers.WriteFirst(tempTransactionLogFilename, new
285         ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
286
287     // Try load damaged database
288     try
289     {
290         // TODO: Fix

```

```

286         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
            ↳ tempTransactionLogFilename))
287     using (var links = new UInt64Links(memoryAdapter))
288     {
289         Global.Trash = links.Count();
290     }
291 }
292 catch (NotSupportedException ex)
293 {
294     Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
        ↳ yet.");
295 }
296
297 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
    ↳ sactionLogFilename);
298
299 File.Delete(tempDatabaseFilename);
300 File.Delete(tempTransactionLogFilename);
301 }
302
303 [Fact]
304 public static void Bug1Test()
305 {
306     var tempDatabaseFilename = Path.GetTempFileName();
307     var tempTransactionLogFilename = Path.GetTempFileName();
308
309     var itself = _constants.Itself;
310
311     // User Code Error (Autoreverted), some data saved
312     try
313     {
314         ulong l1;
315         ulong l2;
316
317         using (var memory = new UInt64ResizableDirectMemoryLinks(tempDatabaseFilename))
318         using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
            ↳ tempTransactionLogFilename))
319         using (var links = new UInt64Links(memoryAdapter))
320         {
321             l1 = links.CreateAndUpdate(itself, itself);
322             l2 = links.CreateAndUpdate(itself, itself);
323
324             l2 = links.Update(l2, l2, l1, l2);
325
326             links.CreateAndUpdate(l2, itself);
327             links.CreateAndUpdate(l2, itself);
328         }
329
330         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
            ↳ TransactionLogFilename);
331
332         using (var memory = new 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
421     private static void DefaultFormatter(StringBuilder sb, ulong link)
422     {
423         sb.Append(link.ToString());
424     }
425
426     #endregion
427
428     #region Performance
429
430     /*
431     public static void RunAllPerformanceTests()
432     {
433         try
434         {
435             links.TestLinksInSteps();
436         }
437     }
438     */

```

```

429     }
430     catch (Exception ex)
431     {
432         ex.WriteToConsole();
433     }
434
435     return;
436
437     try
438     {
439         //ThreadPool.SetMaxThreads(2, 2);
440
441         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
↪ результат
442         // Также это дополнительно помогает в отладке
443         // Увеличивает вероятность попадания информации в кэши
444         for (var i = 0; i < 10; i++)
445         {
446             //0 - 10 ГБ
447             //Каждые 100 МБ срез цифр
448
449             //links.TestGetSourceFunction();
450             //links.TestGetSourceFunctionInParallel();
451             //links.TestGetTargetFunction();
452             //links.TestGetTargetFunctionInParallel();
453             links.Create64BillionLinks();
454
455             links.TestRandomSearchFixed();
456             //links.Create64BillionLinksInParallel();
457             links.TestEachFunction();
458             //links.TestForeach();
459             //links.TestParallelForeach();
460         }
461
462         links.TestDeletionOfAllLinks();
463
464     }
465     catch (Exception ex)
466     {
467         ex.WriteToConsole();
468     }
469 }*/
470
471 /*
472 public static void TestLinksInSteps()
473 {
474     const long gibibyte = 1024 * 1024 * 1024;
475     const long mebibyte = 1024 * 1024;
476
477     var totalLinksToCreate = gibibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478     var linksStep = 102 * mebibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
479
480     var creationMeasurements = new List<TimeSpan>();
481     var 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         Console.WriteLine("\rD {0}/{1}", i + 1, loops);
507     }
508
509     ConsoleHelpers.Debug();
510
511     ConsoleHelpers.Debug("C S D");
512
513     for (int i = 0; i < loops; i++)
514     {
515         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
516 ↪ 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,
524 ↪ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
525     }
526
527     ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
528 ↪ links.Total);
529
530     private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
531 ↪ amountToCreate)
532     {
533         for (long i = 0; i < amountToCreate; i++)
534             links.Create(0, 0);
535     }
536
537     private static TimeSpan GetBaseRandomLoopOverhead(long loops)
538     {
539         return Measure(() =>
540         {
541             ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
542             ulong result = 0;
543             for (long i = 0; i < loops; i++)
544             {
545                 var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
546                 var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
547
548                 result += maxValue + source + target;
549             }
550             Global.Trash = result;
551         });
552     }
553
554     /*
555     [Fact(Skip = "performance test")]
556     public static void GetSourceTest()
557     {
558         using (var scope = new TempLinksTestScope())
559         {
560             var links = scope.Links;
561             ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
562 ↪ Iterations);
563
564             ulong counter = 0;
565
566             //var firstLink = links.First();
567             // Создаём одну связь, из которой будет производить считывание
568             var firstLink = links.Create();
569
570             var sw = Stopwatch.StartNew();
571
572             // Тестируем саму функцию
573             for (ulong i = 0; i < Iterations; i++)
574             {
575                 counter += links.GetSource(firstLink);
576             }
577
578             var elapsedTime = sw.Elapsed;
579
580             var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
581
582             // Удаляем связь, из которой производилось считывание

```

```

580         links.Delete(firstLink);
581
582         ConsoleHelpers.Debug(
583             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
             ↳ second), counter result: {3}",
             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
584     }
585 }
586
587 [Fact(Skip = "performance test")]
588 public static void GetSourceInParallel()
589 {
590     using (var scope = new TempLinksTestScope())
591     {
592         var links = scope.Links;
593         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
594             ↳ parallel.", Iterations);
595
596         long counter = 0;
597
598         //var firstLink = links.First();
599         var firstLink = links.Create();
600
601         var sw = Stopwatch.StartNew();
602
603         // Тестируем саму функцию
604         Parallel.For(0, Iterations, x =>
605         {
606             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
607             //Interlocked.Increment(ref counter);
608         });
609
610         var elapsedTime = sw.Elapsed;
611
612         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
613
614         links.Delete(firstLink);
615
616         ConsoleHelpers.Debug(
617             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
             ↳ second), counter result: {3}",
             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
618     }
619 }
620
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
             ↳ second), counter result: {3}",
             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
650     }
651 }
652
653

```



```

654 [Fact(Skip = "performance test")]
655 public static void TestGetTargetInParallel()
656 {
657     using (var scope = new TempLinksTestScope())
658     {
659         var links = scope.Links;
660         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
        ↳ parallel.", Iterations);
661
662         long counter = 0;
663
664         //var firstLink = links.First();
665         var firstLink = links.Create();
666
667         var sw = Stopwatch.StartNew();
668
669         Parallel.For(0, Iterations, x =>
670         {
671             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
672             //Interlocked.Increment(ref counter);
673         });
674
675         var elapsedTime = sw.Elapsed;
676
677         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
678
679         links.Delete(firstLink);
680
681         ConsoleHelpers.Debug(
682             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
683     }
684 }
685
686 // TODO: Заполнить базу данных перед тестом
687 /*
688 [Fact]
689 public void TestRandomSearchFixed()
690 {
691     var tempFilename = Path.GetTempFileName();
692
693     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
694 ↳ DefaultLinksSizeStep))
695     {
696         long iterations = 64 * 1024 * 1024 /
697 ↳ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
698
699         ulong counter = 0;
700         var maxLink = links.Total;
701
702         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
703
704         var sw = Stopwatch.StartNew();
705
706         for (var i = iterations; i > 0; i--)
707         {
708             var source =
709 ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
710             var target =
711 ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
712
713             counter += links.Search(source, target);
714         }
715
716         var elapsedTime = sw.Elapsed;
717
718         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
719
720         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
721 ↳ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
722 ↳ counter);
723     }
724
725     File.Delete(tempFilename);
726 }*/
727
728 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
729 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.",
736                               ↪ links.Count());
737
738         var sw = Stopwatch.StartNew();
739
740         for (var i = iterations; i > 0; i--)
741         {
742             var linksAddressRange = new
743                 ↪ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
744
745             var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
746             var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
747
748             counter += links.SearchOrDefault(source, target);
749         }
750
751         var elapsedTime = sw.Elapsed;
752
753         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
754
755         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
756                               ↪ Iterations per second), c: {3}",
757                               iterations, elapsedTime, (long)iterationsPerSecond, counter);
758     }
759 }
760
761 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
762 public static void TestEach()
763 {
764     using (var scope = new TempLinksTestScope())
765     {
766         var links = scope.Links;
767
768         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
769
770         ConsoleHelpers.Debug("Testing Each function.");
771
772         var sw = Stopwatch.StartNew();
773
774         links.Each(counter.IncrementAndReturnTrue);
775
776         var elapsedTime = sw.Elapsed;
777
778         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
779
780         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
781                               ↪ links per second)",
782                               counter, elapsedTime, (long)linksPerSecond);
783     }
784 }
785
786 /*
787 [Fact]
788 public static void TestForeach()
789 {
790     var tempFilename = Path.GetTempFileName();
791
792     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
793 ↪ DefaultLinksSizeStep))
794     {
795         ulong counter = 0;
796
797         ConsoleHelpers.Debug("Testing foreach through links.");
798
799         var sw = Stopwatch.StartNew();
800
801         //foreach (var link in links)
802         //{
803             //    counter++;
804         //}

```

```

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.Doublents.Links(tempFilename,
↪ DefaultLinksSizeStep))
818     {
819         long counter = 0;
820
821         ConsoleHelpers.Debug("Testing parallel foreach through links.");
822
823         var sw = Stopwatch.StartNew();
824
825         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
826         //{
827         //    Interlocked.Increment(ref counter);
828         //});
829
830         var elapsedTime = sw.Elapsed;
831
832         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
833
834         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
835     }
836
837     File.Delete(tempFilename);
838 }
839 */
840
841 [Fact(Skip = "performance test")]
842 public static void Create64BillionLinks()
843 {
844     using (var scope = new TempLinksTestScope())
845     {
846         var links = scope.Links;
847         var linksBeforeTest = links.Count();
848
849         long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
850
851         ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
852
853         var elapsedTime = Performance.Measure(() =>
854         {
855             for (long i = 0; i < linksToCreate; i++)
856             {
857                 links.Create();
858             }
859         });
860
861         var linksCreated = links.Count() - linksBeforeTest;
862         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
863
864         ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
865
866         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↪ linksCreated, elapsedTime,
867             (long)linksPerSecond);
868     }
869 }
870
871 [Fact(Skip = "performance test")]
872
873

```



```

26         unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
27     }
28     var fromUnaryNumberConverterUsingOrOperation = new
    ↪     UnaryNumberToAddressOrOperationConverter<ulong>(links,
    ↪     powerOf2ToUnaryNumberConverter);
29     var fromUnaryNumberConverterUsingAddOperation = new
    ↪     UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
30     for (int i = 0; i < N; i++)
31     {
32         Assert.Equal(numbers[i],
    ↪         fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
33         Assert.Equal(numbers[i],
    ↪         fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
34     }
35 }
36 }
37 }
38 }

```

1.115 ./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,
    ↪ frequencyPropertyOperator, frequencyIncrementer);
85         var linkToItsFrequencyNumberConverter = new
    ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
    ↪ unaryNumberToAddressConverter);
86         var sequenceToItsLocalElementLevelsConverter = new
    ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
    ↪ linkToItsFrequencyNumberConverter);
87         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↪ sequenceToItsLocalElementLevelsConverter);
88
89         var stringToUnicodeSequenceConverter = new
    ↪ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
    ↪ index, optimalVariantConverter, unicodeSequenceMarker);
90
91         var originalString = "Hello";
92
93         var unicodeSequenceLink =
    ↪ stringToUnicodeSequenceConverter.Convert(originalString);
94
95         var unicodeSymbolCriterionMatcher = new
    ↪ UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
96         var unicodeSymbolToCharConverter = new
    ↪ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
    ↪ unicodeSymbolCriterionMatcher);
97
98         var unicodeSequenceCriterionMatcher = new
    ↪ UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
99
100        var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
    ↪ unicodeSymbolCriterionMatcher.IsMatched);

```

```
102         var unicodeSequenceToStringConverter = new
            ↳ UnicodeSequenceToStringConverter<ulong>(links,
            ↳ unicodeSequenceCriterionMatcher, sequenceWalker,
            ↳ unicodeSymbolToCharConverter);
103
104         var resultingString =
            ↳ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
105
106         Assert.Equal(originalString, resultingString);
107     }
108 }
109 }
110 }
```

Index

- ./Platform.Data.Doublets.Tests/ComparisonTests.cs, 149
- ./Platform.Data.Doublets.Tests/EqualityTests.cs, 150
- ./Platform.Data.Doublets.Tests/GenericLinksTests.cs, 151
- ./Platform.Data.Doublets.Tests/LinksConstantsTests.cs, 152
- ./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 152
- ./Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 155
- ./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 156
- ./Platform.Data.Doublets.Tests/ScopeTests.cs, 157
- ./Platform.Data.Doublets.Tests/SequencesTests.cs, 158
- ./Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 172
- ./Platform.Data.Doublets.Tests/TestExtensions.cs, 173
- ./Platform.Data.Doublets.Tests/UInt64LinksTests.cs, 176
- ./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 188
- ./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 189
- ./Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs, 1
- ./Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs, 1
- ./Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs, 1
- ./Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs, 2
- ./Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs, 3
- ./Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs, 4
- ./Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs, 4
- ./Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs, 5
- ./Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs, 5
- ./Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs, 5
- ./Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs, 6
- ./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs, 6
- ./Platform.Data.Doublets/Decorators/UInt64Links.cs, 6
- ./Platform.Data.Doublets/Decorators/UniLinks.cs, 7
- ./Platform.Data.Doublets/Doublet.cs, 12
- ./Platform.Data.Doublets/DoubletComparer.cs, 13
- ./Platform.Data.Doublets/ILinks.cs, 13
- ./Platform.Data.Doublets/ILinksExtensions.cs, 13
- ./Platform.Data.Doublets/ISynchronizedLinks.cs, 25
- ./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs, 25
- ./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs, 26
- ./Platform.Data.Doublets/Link.cs, 26
- ./Platform.Data.Doublets/LinkExtensions.cs, 29
- ./Platform.Data.Doublets/LinksOperatorBase.cs, 30
- ./Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 30
- ./Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConverter.cs, 30
- ./Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 31
- ./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 32
- ./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 33
- ./Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 34
- ./Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 34
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksAvlBalancedTreeMethodsBase.cs, 35
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSizeBalancedTreeMethodsBase.cs, 39
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesAvlBalancedTreeMethods.cs, 43
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesSizeBalancedTreeMethods.cs, 44
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsAvlBalancedTreeMethods.cs, 45
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsSizeBalancedTreeMethods.cs, 46
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinks.cs, 47
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinksBase.cs, 48
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/UnusedLinksListMethods.cs, 55
- ./Platform.Data.Doublets/ResizableDirectMemory/ILinksListMethods.cs, 56
- ./Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.cs, 56
- ./Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs, 57
- ./Platform.Data.Doublets/ResizableDirectMemory/RawLink.cs, 57
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs, 58
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs, 60
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs, 61
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs, 62
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 63
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 64
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64ResizableDirectMemoryLinks.cs, 65
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64UnusedLinksListMethods.cs, 67

./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 67
./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 68
./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 71
./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 72
./Platform.Data.Doublets/Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 73
./Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs, 74
./Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs, 74
./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 74
./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 75
./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 75
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 78
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 80
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 80
./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 80
./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 81
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 82
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 82
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 82
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 83
./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 84
./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 84
./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 85
./Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 85
./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 86
./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 87
./Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 87
./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 87
./Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 88
./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 88
./Platform.Data.Doublets/Sequences/Sequences.cs, 116
./Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 126
./Platform.Data.Doublets/Sequences/SequencesOptions.cs, 127
./Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 130
./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 130
./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 130
./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 132
./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 133
./Platform.Data.Doublets/Stacks/Stack.cs, 134
./Platform.Data.Doublets/Stacks/StackExtensions.cs, 134
./Platform.Data.Doublets/SynchronizedLinks.cs, 135
./Platform.Data.Doublets/UInt64LinksExtensions.cs, 136
./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 138
./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 143
./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 144
./Platform.Data.Doublets/Unicode/UnicodeMap.cs, 144
./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 147
./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 147
./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 148
./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 148