

LinksPlatform's Platform.Data.Doublets Class Library

./Converters/AddressToUnaryNumberConverter.cs

```
1 using System.Collections.Generic;
2 using Platform.Interfaces;
3 using Platform.Reflection;
4 using Platform.Numbers;
5
6 namespace Platform.Data.Doublets.Converters
7 {
8     public class AddressToUnaryNumberConverter<TLink> :
9         ↳ LinksOperatorBase<TLink>, IConverter<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↳ EqualityComparer<TLink>.Default;
13
14         private readonly IConverter<int, TLink>
15             ↳ _powerOf2ToUnaryNumberConverter;
16
17         public AddressToUnaryNumberConverter(ILinks<TLink> links,
18             ↳ IConverter<int, TLink> powerOf2ToUnaryNumberConverter) :
19             ↳ base(links) => _powerOf2ToUnaryNumberConverter =
20             ↳ powerOf2ToUnaryNumberConverter;
21
22         public TLink Convert(TLink sourceAddress)
23         {
24             var number = sourceAddress;
25             var target = Links.Constants.Null;
26             for (int i = 0; i < CachedTypeInfo<TLink>.BitsLength; i++)
27             {
28                 if (_equalityComparer.Equals(ArithmeticHelpers.And(number,
29                     ↳ Integer<TLink>.One), Integer<TLink>.One))
30                 {
31                     target = _equalityComparer.Equals(target,
32                         ↳ Links.Constants.Null)
33                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
34                         : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Co
35                             ↳ nvert(i),
36                             ↳ target);
37                 }
38                 number = (Integer<TLink>)((ulong)(Integer<TLink>)number >> 1);
39                 ↳ // Should be BitwiseHelpers.ShiftRight(number, 1);
40                 if (_equalityComparer.Equals(number, default))
41                 {
42                     break;
43                 }
44             }
45             return target;
46         }
47     }
48 }
```

./Converters/LinkToItsFrequencyNumberConveter.cs

```
1 using System;
2 using System.Collections.Generic;
3 using Platform.Interfaces;
4
5 namespace Platform.Data.Doublets.Converters
6 {
7     public class LinkToItsFrequencyNumberConveter<TLink> :
8         ↳ LinksOperatorBase<TLink>, IConverter<Doublet<TLink>, TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↳ EqualityComparer<TLink>.Default;
```

```
11 private readonly ISpecificPropertyOperator<TLink, TLink>
12     ↳ _frequencyPropertyOperator;
13 private readonly IConverter<TLink> _unaryNumberToAddressConverter;
14
15 public LinkToItsFrequencyNumberConveter(
16     ILinks<TLink> links,
17     ISpecificPropertyOperator<TLink, TLink> frequencyPropertyOperator,
18     IConverter<TLink> unaryNumberToAddressConverter)
19     : base(links)
20 {
21     _frequencyPropertyOperator = frequencyPropertyOperator;
22     _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
23 }
24
25 public TLink Convert(Doublet<TLink> doublet)
26 {
27     var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
28     if (_equalityComparer.Equals(link, Links.Constants.Null))
29     {
30         throw new ArgumentException($"Link with {doublet.Source}
31             ↳ source and {doublet.Target} target not found.",
32             ↳ nameof(doublet));
33     }
34     var frequency = _frequencyPropertyOperator.Get(link);
35     if (_equalityComparer.Equals(frequency, default))
36     {
37         return default;
38     }
39     var frequencyNumber = Links.GetSource(frequency);
40     var number =
41         ↳ _unaryNumberToAddressConverter.Convert(frequencyNumber);
42     return number;
43 }
44 }
```

./Converters/PowerOf2ToUnaryNumberConverter.cs

```
1 using System;
2 using System.Collections.Generic;
3 using Platform.Interfaces;
4
5 namespace Platform.Data.Doublets.Converters
6 {
7     public class PowerOf2ToUnaryNumberConverter<TLink> :
8         ↳ LinksOperatorBase<TLink>, IConverter<int, TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↳ EqualityComparer<TLink>.Default;
12
13         private readonly TLink[] _unaryNumberPowersOf2;
14
15         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one)
16             ↳ : base(links)
17         {
18             _unaryNumberPowersOf2 = new TLink[64];
19             _unaryNumberPowersOf2[0] = one;
20         }
21
22         public TLink Convert(int power)
23         {
24             if (power < 0 || power >= _unaryNumberPowersOf2.Length)
25             {
26                 throw new ArgumentOutOfRangeException(nameof(power));
27             }
28         }
29     }
```



```

38     }
39     target = (Integer<TLink>)((Integer<TLink>)target | 1UL <<
        ↳ powerOf2Index); // MathHelpers.Or(target,
        ↳ MathHelpers.ShiftLeft(One, powerOf2Index))
40 }
41 return target;
42 }
43 }
44 }

```

./Decorators/LinksCascadeDependenciesResolver.cs

```

1 using System.Collections.Generic;
2 using Platform.Collections.Arrays;
3 using Platform.Numbers;
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public class LinksCascadeDependenciesResolver<TLink> :
        ↳ LinksDecoratorBase<TLink>
8     {
9         private static readonly EqualityComparer<TLink> _equalityComparer =
        ↳ EqualityComparer<TLink>.Default;
10
11         public LinksCascadeDependenciesResolver(ILinks<TLink> links) :
        ↳ base(links) { }
12
13         public override void Delete(TLink link)
14         {
15             EnsureNoDependenciesOnDelete(link);
16             base.Delete(link);
17         }
18
19         public void EnsureNoDependenciesOnDelete(TLink link)
20         {
21             ulong referencesCount = (Integer<TLink>)Links.Count(Constants.Any,
        ↳ link);
22             var references = ArrayPool.Allocate<TLink>((long)referencesCount);
23             var referencesFiller = new ArrayFiller<TLink, TLink>(references,
        ↳ Constants.Continue);
24             Links.Each(referencesFiller.AddFirstAndReturnConstant,
        ↳ Constants.Any, link);
25             //references.Sort() // TODO: Решить необходимо ли для корректного
        ↳ порядка отмены операций в транзакциях
26             for (var i = (long)referencesCount - 1; i >= 0; i--)
27             {
28                 if (_equalityComparer.Equals(references[i], link))
29                 {
30                     continue;
31                 }
32                 Links.Delete(references[i]);
33             }
34             ArrayPool.Free(references);
35         }
36     }
37 }

```

./Decorators/LinksCascadeUniquenessAndDependenciesResolver.cs

```

1 using System.Collections.Generic;
2 using Platform.Collections.Arrays;
3 using Platform.Numbers;
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public class LinksCascadeUniquenessAndDependenciesResolver<TLink> :
        ↳ LinksUniquenessResolver<TLink>
8     {

```

```

9         private static readonly EqualityComparer<TLink> _equalityComparer =
        ↳ EqualityComparer<TLink>.Default;
10
11         public LinksCascadeUniquenessAndDependenciesResolver(ILinks<TLink>
        ↳ links) : base(links) { }
12
13         protected override TLink ResolveAddressChangeConflict(TLink
        ↳ oldLinkAddress, TLink newLinkAddress)
14         {
15             // TODO: Very similar to Merge (logic should be reused)
16             ulong referencesAsSourceCount =
        ↳ (Integer<TLink>)Links.Count(Constants.Any, oldLinkAddress,
        ↳ Constants.Any);
17             ulong referencesAsTargetCount =
        ↳ (Integer<TLink>)Links.Count(Constants.Any, Constants.Any,
        ↳ oldLinkAddress);
18             var references =
        ↳ ArrayPool.Allocate<TLink>((long)(referencesAsSourceCount +
        ↳ referencesAsTargetCount));
19             var referencesFiller = new ArrayFiller<TLink, TLink>(references,
        ↳ Constants.Continue);
20             Links.Each(referencesFiller.AddFirstAndReturnConstant,
        ↳ Constants.Any, oldLinkAddress, Constants.Any);
21             Links.Each(referencesFiller.AddFirstAndReturnConstant,
        ↳ Constants.Any, Constants.Any, oldLinkAddress);
22             for (ulong i = 0; i < referencesAsSourceCount; i++)
23             {
24                 var reference = references[i];
25                 if (!_equalityComparer.Equals(reference, oldLinkAddress))
26                 {
27                     Links.Update(reference, newLinkAddress,
        ↳ Links.GetTarget(reference));
28                 }
29             }
30             for (var i = (long)referencesAsSourceCount; i < references.Length;
        ↳ i++)
31             {
32                 var reference = references[i];
33                 if (!_equalityComparer.Equals(reference, oldLinkAddress))
34                 {
35                     Links.Update(reference, Links.GetSource(reference),
        ↳ newLinkAddress);
36                 }
37             }
38             ArrayPool.Free(references);
39             return base.ResolveAddressChangeConflict(oldLinkAddress,
        ↳ newLinkAddress);
40         }
41     }
42 }

```

./Decorators/LinksDecoratorBase.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Data.Constants;
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public abstract class LinksDecoratorBase<T> : ILinks<T>
8     {
9         public LinksCombinedConstants<T, T, int> Constants { get; }
10
11         public readonly ILinks<T> Links;

```

```

12
13     protected LinksDecoratorBase(ILinks<T> links)
14     {
15         Links = links;
16         Constants = links.Constants;
17     }
18
19     public virtual T Count(IList<T> restriction) =>
20         ↳ Links.Count(restriction);
21
22     public virtual T Each(Func<IList<T>, T> handler, IList<T>
23         ↳ restrictions) => Links.Each(handler, restrictions);
24
25     public virtual T Create() => Links.Create();
26
27     public virtual T Update(IList<T> restrictions) =>
28         ↳ Links.Update(restrictions);
29
30     public virtual void Delete(T link) => Links.Delete(link);
31 }
32 }

```

./Decorators/LinksDependenciesValidator.cs

```

1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Decorators
4 {
5     public class LinksDependenciesValidator<T> : LinksDecoratorBase<T>
6     {
7         public LinksDependenciesValidator(ILinks<T> links) : base(links) { }
8
9         public override T Update(IList<T> restrictions)
10         {
11             Links.EnsureNoDependencies(restrictions[Constants.IndexPart]);
12             return base.Update(restrictions);
13         }
14
15         public override void Delete(T link)
16         {
17             Links.EnsureNoDependencies(link);
18             base.Delete(link);
19         }
20     }
21 }

```

./Decorators/LinksDisposableDecoratorBase.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Disposables;
4 using Platform.Data.Constants;
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     public abstract class LinksDisposableDecoratorBase<T> : DisposableBase,
9         ↳ ILinks<T>
10     {
11         public LinksCombinedConstants<T, T, int> Constants { get; }
12
13         public readonly ILinks<T> Links;
14
15         protected LinksDisposableDecoratorBase(ILinks<T> links)
16         {
17             Links = links;
18             Constants = links.Constants;
19         }
20     }
21 }

```

```

20     public virtual T Count(IList<T> restriction) =>
21         ↳ Links.Count(restriction);
22
23     public virtual T Each(Func<IList<T>, T> handler, IList<T>
24         ↳ restrictions) => Links.Each(handler, restrictions);
25
26     public virtual T Create() => Links.Create();
27
28     public virtual T Update(IList<T> restrictions) =>
29         ↳ Links.Update(restrictions);
30
31     public virtual void Delete(T link) => Links.Delete(link);
32
33     protected override bool AllowMultipleDisposeCalls => true;
34
35     protected override void DisposeCore(bool manual, bool wasDisposed) =>
36         ↳ Disposable.TryDispose(Links);
37 }
38 }

```

./Decorators/LinksInnerReferenceValidator.cs

```

1 using System;
2 using System.Collections.Generic;
3
4 namespace Platform.Data.Doublets.Decorators
5 {
6     // TODO: Make LinksExternalReferenceValidator. A layer that checks each
7     ↳ link to exist or to be external (hybrid link's raw number).
8     public class LinksInnerReferenceValidator<T> : LinksDecoratorBase<T>
9     {
10         public LinksInnerReferenceValidator(ILinks<T> links) : base(links) { }
11
12         public override T Each(Func<IList<T>, T> handler, IList<T>
13             ↳ restrictions)
14         {
15             Links.EnsureInnerReferenceExists(restrictions,
16                 ↳ nameof(restrictions));
17             return base.Each(handler, restrictions);
18         }
19
20         public override T Count(IList<T> restriction)
21         {
22             Links.EnsureInnerReferenceExists(restriction, nameof(restriction));
23             return base.Count(restriction);
24         }
25
26         public override T Update(IList<T> restrictions)
27         {
28             // TODO: Possible values: null, ExistentLink or
29             ↳ NonExistentHybrid(ExternalReference)
30             Links.EnsureInnerReferenceExists(restrictions,
31                 ↳ nameof(restrictions));
32             return base.Update(restrictions);
33         }
34
35         public override void Delete(T link)
36         {
37             // TODO: Решить считать ли такое исключением, или лишь более
38             ↳ конкретным требованием?
39             Links.EnsureLinkExists(link, nameof(link));
40             base.Delete(link);
41         }
42     }
43 }

```

./Decorators/LinksNonExistentReferencesCreator.cs

```
1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Decorators
4 {
5     /// <remarks>
6     /// Not practical if newSource and newTarget are too big.
7     /// To be able to use practical version we should allow to create link at
8     /// ↪ any specific location inside ResizableDirectMemoryLinks.
9     /// This in turn will require to implement not a list of empty links, but
10    /// ↪ a list of ranges to store it more efficiently.
11    /// </remarks>
12    public class LinksNonExistentReferencesCreator<T> : LinksDecoratorBase<T>
13    {
14        public LinksNonExistentReferencesCreator(ILinks<T> links) :
15            ↪ base(links) { }
16
17        public override T Update(IList<T> restrictions)
18        {
19            Links.EnsureCreated(restrictions[Constants.SourcePart],
20            ↪ restrictions[Constants.TargetPart]);
21            return base.Update(restrictions);
22        }
23    }
24 }
```

./Decorators/LinksNullToSelfReferenceResolver.cs

```
1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Decorators
4 {
5     public class LinksNullToSelfReferenceResolver<TLink> :
6     ↪ LinksDecoratorBase<TLink>
7     {
8         private static readonly EqualityComparer<TLink> _equalityComparer =
9         ↪ EqualityComparer<TLink>.Default;
10
11        public LinksNullToSelfReferenceResolver(ILinks<TLink> links) :
12        ↪ base(links) { }
13
14        public override TLink Create()
15        {
16            var link = base.Create();
17            return Links.Update(link, link, link);
18        }
19
20        public override TLink Update(IList<TLink> restrictions)
21        {
22            restrictions[Constants.SourcePart] =
23            ↪ _equalityComparer.Equals(restrictions[Constants.SourcePart],
24            ↪ Constants.Null) ? restrictions[Constants.SourcePart] :
25            ↪ restrictions[Constants.SourcePart];
26            restrictions[Constants.TargetPart] =
27            ↪ _equalityComparer.Equals(restrictions[Constants.TargetPart],
28            ↪ Constants.Null) ? restrictions[Constants.TargetPart] :
29            ↪ restrictions[Constants.TargetPart];
30            return base.Update(restrictions);
31        }
32    }
33 }
```

./Decorators/LinksSelfReferenceResolver.cs

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

```
4 namespace Platform.Data.Doublets.Decorators
5 {
6     public class LinksSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
7     {
8         private static readonly EqualityComparer<TLink> _equalityComparer =
9         ↪ EqualityComparer<TLink>.Default;
10
11        public LinksSelfReferenceResolver(ILinks<TLink> links) : base(links) {
12        ↪ }
13
14        public override TLink Each(Func<IList<TLink>, TLink> handler,
15        ↪ IList<TLink> restrictions)
16        {
17            if (!_equalityComparer.Equals(Constants.Any, Constants.Itself)
18            && ((restrictions.Count > Constants.IndexPart) &&
19            ↪ _equalityComparer.Equals(restrictions[Constants.IndexPart],
20            ↪ Constants.Itself))
21            || ((restrictions.Count > Constants.SourcePart) &&
22            ↪ _equalityComparer.Equals(restrictions[Constants.SourcePart],
23            ↪ Constants.Itself))
24            || ((restrictions.Count > Constants.TargetPart) &&
25            ↪ _equalityComparer.Equals(restrictions[Constants.TargetPart],
26            ↪ Constants.Itself))))
27            {
28                return Constants.Continue;
29            }
30            return base.Each(handler, restrictions);
31        }
32
33        public override TLink Update(IList<TLink> restrictions)
34        {
35            restrictions[Constants.SourcePart] =
36            ↪ _equalityComparer.Equals(restrictions[Constants.SourcePart],
37            ↪ Constants.Itself) ? restrictions[Constants.SourcePart] :
38            ↪ restrictions[Constants.SourcePart];
39            restrictions[Constants.TargetPart] =
40            ↪ _equalityComparer.Equals(restrictions[Constants.TargetPart],
41            ↪ Constants.Itself) ? restrictions[Constants.TargetPart] :
42            ↪ restrictions[Constants.TargetPart];
43            return base.Update(restrictions);
44        }
45    }
46 }
```

./Decorators/LinksUniquenessResolver.cs

```
1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Decorators
4 {
5     public class LinksUniquenessResolver<TLink> : LinksDecoratorBase<TLink>
6     {
7         private static readonly EqualityComparer<TLink> _equalityComparer =
8         ↪ EqualityComparer<TLink>.Default;
9
10        public LinksUniquenessResolver(ILinks<TLink> links) : base(links) { }
11
12        public override TLink Update(IList<TLink> restrictions)
13        {
14            var newLinkAddress =
15            ↪ Links.SearchOrDefault(restrictions[Constants.SourcePart],
16            ↪ restrictions[Constants.TargetPart]);
17            if (_equalityComparer.Equals(newLinkAddress, default))
18            {
19                return base.Update(restrictions);
20            }
21        }
22    }
23 }
```

```

18         return ResolveAddressChangeConflict(restrictions[Constants.IndexPa
    ↪ rt],
    ↪ newLinkAddress);
19     }
20
21     protected virtual TLink ResolveAddressChangeConflict(TLink
    ↪ oldLinkAddress, TLink newLinkAddress)
22     {
23         if (Links.Exists(oldLinkAddress))
24         {
25             Delete(oldLinkAddress);
26         }
27         return newLinkAddress;
28     }
29 }
30 }

```

./Decorators/LinksUniquenessValidator.cs

```

1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Decorators
4 {
5     public class LinksUniquenessValidator<T> : LinksDecoratorBase<T>
6     {
7         public LinksUniquenessValidator(ILinks<T> links) : base(links) { }
8
9         public override T Update(IList<T> restrictions)
10        {
11            Links.EnsureDoesNotExists(restrictions[Constants.SourcePart],
    ↪ restrictions[Constants.TargetPart]);
12            return base.Update(restrictions);
13        }
14    }
15 }

```

./Decorators/NonNullContentsLinkDeletionResolver.cs

```

1 namespace Platform.Data.Doublets.Decorators
2 {
3     public class NonNullContentsLinkDeletionResolver<T> : LinksDecoratorBase<T>
4     {
5         public NonNullContentsLinkDeletionResolver(ILinks<T> links) :
    ↪ base(links) { }
6
7         public override void Delete(T link)
8         {
9             Links.Update(link, Constants.Null, Constants.Null);
10            base.Delete(link);
11        }
12    }
13 }

```

./Decorators/UInt64Links.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Collections;
4 using Platform.Collections.Arrays;
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     /// <summary>
9     /// Представляет объект для работы с базой данных (файлом) в формате Links
    ↪ (массива взаимосвязей).
10    /// </summary>
11    /// <remarks>
12    /// Возможные оптимизации:

```

```

13    /// Объединение в одном 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    /// AVL - высота дерева может позволить точно рассчитать размер дерева,
    ↪ нет необходимости в SBT.
22    /// AVL дерево можно прошить.
23    ///
24    /// Текущее теоретическое ограничение на размер связей - long.MaxValue
25    /// Желательно реализовать поддержку переключения между деревьями и
    ↪ битовыми индексами (битовыми строками) - вариант матрицы
    ↪ (выстраиваемой лениво).
26    ///
27    /// Решить отключать ли проверки при компиляции под Release. Т.е.
    ↪ исключения будут выбрасываться только при #if DEBUG
28    /// </remarks>
29    public class UInt64Links : LinksDisposableDecoratorBase<ulong>
30    {
31        public UInt64Links(ILinks<ulong> links) : base(links) { }
32
33        public override ulong Each(Func<IList<ulong>, ulong> handler,
    ↪ IList<ulong> restrictions)
34        {
35            this.EnsureLinkIsAnyOrExists(restrictions);
36            return Links.Each(handler, restrictions);
37        }
38
39        public override ulong Create() => Links.CreatePoint();
40
41        public override ulong Update(IList<ulong> restrictions)
42        {
43            if (restrictions.IsNullOrEmpty())
44            {
45                return Constants.Null;
46            }
47            // TODO: Remove usages of these hacks (these should not be
    ↪ backwards compatible)
48            if (restrictions.Count == 2)
49            {
50                return this.Merge(restrictions[0], restrictions[1]);
51            }
52            if (restrictions.Count == 4)
53            {
54                return this.UpdateOrCreateOrGet(restrictions[0],
    ↪ restrictions[1], restrictions[2], restrictions[3]);
55            }
56            // TODO: Looks like this is a common type of exceptions linked
    ↪ with restrictions support
57            if (restrictions.Count != 3)
58            {
59                throw new NotSupportedException();
60            }
61            var updatedLink = restrictions[Constants.IndexPart];
62            this.EnsureLinkExists(updatedLink, nameof(Constants.IndexPart));
63            var newSource = restrictions[Constants.SourcePart];
64            this.EnsureLinkIsItselfOrExists(newSource,
    ↪ nameof(Constants.SourcePart));
65            var newTarget = restrictions[Constants.TargetPart];
66            this.EnsureLinkIsItselfOrExists(newTarget,
    ↪ nameof(Constants.TargetPart));

```

```

67     var existedLink = Constants.Null;
68     if (newSource != Constants.Itself && newTarget != Constants.Itself)
69     {
70         existedLink = this.SearchOrDefault(newSource, newTarget);
71     }
72     if (existedLink == Constants.Null)
73     {
74         var before = Links.GetLink(updatedLink);
75         if (before[Constants.SourcePart] != newSource ||
76             ↪ before[Constants.TargetPart] != newTarget)
77         {
78             Links.Update(updatedLink, newSource == Constants.Itself ?
79                 ↪ updatedLink : newSource,
80                 newTarget == Constants.Itself ?
81                 ↪ updatedLink : newTarget);
82         }
83         return updatedLink;
84     }
85     else
86     {
87         // Replace one link with another (replaced link is deleted,
88         ↪ children are updated or deleted), it is actually merge
89         ↪ operation
90         return this.Merge(updatedLink, existedLink);
91     }
92 }
93
94 /// <summary>Удаляет связь с указанным индексом.</summary>
95 /// <param name="link">Индекс удаляемой связи.</param>
96 public override void Delete(ulong link)
97 {
98     this.EnsureLinkExists(link);
99     Links.Update(link, Constants.Null, Constants.Null);
100     var referencesCount = Links.Count(Constants.Any, link);
101     if (referencesCount > 0)
102     {
103         var references = new ulong[referencesCount];
104         var referencesFiller = new ArrayFiller<ulong>,
105             ↪ ulong>(references, Constants.Continue);
106         Links.Each(referencesFiller.AddFirstAndReturnConstant,
107             ↪ Constants.Any, link);
108         //references.Sort(); // TODO: Решить необходимо ли для
109         ↪ корректного порядка отмены операций в транзакциях
110         for (var i = (long)referencesCount - 1; i >= 0; i--)
111         {
112             if (this.Exists(references[i]))
113             {
114                 Delete(references[i]);
115             }
116         }
117     }
118     //else
119     // TODO: Определить почему здесь есть связи, которых не
120     ↪ существует
121 }
122 Links.Delete(link);
123 }
124 }
125 }

```

./Decorators/UniLinks.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using Platform.Collections;
5 using Platform.Collections.Arrays;
6 using Platform.Collections.Lists;

```

```

7 using Platform.Helpers.Scopes;
8 using Platform.Data.Constants;
9 using Platform.Data.Universal;
10 using System.Collections.ObjectModel;
11
12 namespace Platform.Data.Doublets.Decorators
13 {
14     /// <remarks>
15     /// What does empty pattern (for condition or substitution) mean? Nothing
16     ↪ or Everything?
17     /// Now we go with nothing. And nothing is something one, but empty, and
18     ↪ cannot be changed by itself. But can cause creation (update from
19     ↪ nothing) or deletion (update to nothing).
20     ///
21     /// TODO: Decide to change to IDoubletLinks or not to change. (Better to
22     ↪ create DefaultUniLinksBase, that contains logic itself and can be
23     ↪ implemented using both IDoubletLinks and ILinks.)
24     /// </remarks>
25     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>,
26     ↪ IUniLinks<TLink>
27     {
28         private static readonly EqualityComparer<TLink> _equalityComparer =
29         ↪ EqualityComparer<TLink>.Default;
30
31         public UniLinks(ILinks<TLink> links) : base(links) { }
32
33         private struct Transition
34         {
35             public IList<TLink> Before;
36             public IList<TLink> After;
37
38             public Transition(IList<TLink> before, IList<TLink> after)
39             {
40                 Before = before;
41                 After = after;
42             }
43         }
44
45         public static readonly TLink NullConstant =
46         ↪ Use<LinksCombinedConstants<TLink, TLink, int>>.Single.Null;
47         public static readonly IReadOnlyList<TLink> NullLink = new
48         ↪ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant,
49         ↪ NullConstant, NullConstant });
50
51         // TODO: Подумать о том, как реализовать древовидный Restriction и
52         ↪ Substitution (Links-Expression)
53         public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>,
54         ↪ IList<TLink>, TLink> matchedHandler, IList<TLink> substitution,
55         ↪ Func<IList<TLink>, IList<TLink>, TLink> substitutedHandler)
56         {
57             ///List<Transition> transitions = null;
58             ///if (!restriction.IsNullOrEmpty())
59             ///{
60                 /// // Есть причина делать проход (чтение)
61                 /// if (matchedHandler != null)
62                 /// {
63                     /// if (!substitution.IsNullOrEmpty())
64                     /// {
65                         /// // restriction => { 0, 0, 0 } | { 0 } // Create
66                         /// // substitution => { itself, 0, 0 } | { itself,
67                         ↪ itself, itself } // Create / Update
68                         /// // substitution => { 0, 0, 0 } | { 0 } // Delete
69                         /// transitions = new List<Transition>();
70                         /// if (Equals(substitution[Constants.IndexPart],
71                         ↪ Constants.Null))

```

```

57     /// {
58     ///     // If index is Null, that means we always
59     // ignore every other value (they are also Null by definition)
60     ///     var matchDecision = matchedHandler(, NullLink);
61     ///     if (Equals(matchDecision, Constants.Break))
62     ///         return false;
63     ///     if (!Equals(matchDecision, Constants.Skip))
64     ///         transitions.Add(new
65     // Transition(matchedLink, newValue));
66     ///     }
67     ///     else
68     ///     {
69     ///         Func<T, bool> handler;
70     ///         handler = link =>
71     ///         {
72     ///             var matchedLink =
73     // Memory.GetLinkValue(link);
74     ///             var newValue = Memory.GetLinkValue(link);
75     ///             newValue[Constants.IndexPart] =
76     // Constants.Itself;
77     ///             newValue[Constants.SourcePart] =
78     // Equals(substitution[Constants.SourcePart], Constants.Itself) ?
79     // matchedLink[Constants.IndexPart] :
80     // substitution[Constants.SourcePart];
81     ///             newValue[Constants.TargetPart] =
82     // Equals(substitution[Constants.TargetPart], Constants.Itself) ?
83     // matchedLink[Constants.IndexPart] :
84     // substitution[Constants.TargetPart];
85     ///             var matchDecision =
86     // matchedHandler(matchedLink, newValue);
87     ///             if (Equals(matchDecision, Constants.Break))
88     ///                 return false;
89     ///             if (!Equals(matchDecision, Constants.Skip))
90     ///                 transitions.Add(new
91     // Transition(matchedLink, newValue));
92     ///             return true;
93     ///         };
94     ///         if (!Memory.Each(handler, restriction))
95     ///             return Constants.Break;
96     ///     }
97     /// }
98     /// else
99     /// {
100     // if (substitution != null)
101     // {
102     //     transitions = new List<IList<T>>();
103     //     Func<T, bool> handler = link =>
104     //     {
105     //         var matchedLink = Memory.GetLinkValue(link);
106     //         transitions.Add(matchedLink);
107     //         return true;

```

```

108     ///     };
109     // if (!Memory.Each(handler, restriction))
110     //     return Constants.Break;
111     /// }
112     /// else
113     /// {
114     //     return Constants.Continue;
115     /// }
116     /// }
117     /// }
118     /// if (substitution != null)
119     /// {
120     //     // Есть причина делать замену (запись)
121     //     if (substitutedHandler != null)
122     //     {
123     //     }
124     //     else
125     //     {
126     //     }
127     /// }
128     /// return Constants.Continue;
129
130     // if (restriction.IsNullOrEmpty()) // Create
131     // {
132     //     substitution[Constants.IndexPart] = Memory.AllocateLink();
133     //     Memory.SetLinkValue(substitution);
134     // }
135     // else if (substitution.IsNullOrEmpty()) // Delete
136     // {
137     //     Memory.FreeLink(restriction[Constants.IndexPart]);
138     // }
139     // else if (restriction.EqualTo(substitution)) // Read or ("repeat"
140     // the state) // Each
141     // {
142     //     // No need to collect links to list
143     //     // Skip == Continue
144     //     // No need to check substitutedHandler
145     //     if (!Memory.Each(link =>
146     //         !Equals(matchedHandler(Memory.GetLinkValue(link)),
147     //         Constants.Break), restriction))
148     //         return Constants.Break;
149     // }
150     // else // Update
151     // {
152     //     // List<IList<T>> matchedLinks = null;
153     //     if (matchedHandler != null)
154     //     {
155     //         matchedLinks = new List<IList<T>>();
156     //         Func<T, bool> handler = link =>
157     //         {
158     //             var matchedLink = Memory.GetLinkValue(link);
159     //             var matchDecision = matchedHandler(matchedLink);
160     //             if (Equals(matchDecision, Constants.Break))
161     //                 return false;
162     //             if (!Equals(matchDecision, Constants.Skip))
163     //                 matchedLinks.Add(matchedLink);
164     //             return true;
165     //         };
166     //         if (!Memory.Each(handler, restriction))
167     //             return Constants.Break;
168     //     }
169     //     if (!matchedLinks.IsNullOrEmpty())
170     //     {
171     //         var totalMatchedLinks = matchedLinks.Count;
172     //         for (var i = 0; i < totalMatchedLinks; i++)

```



```

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

```

```

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

```

```

282         Links.Update(linkToUpdate, Constants.Null,
283             ↪ Constants.Null);
284         Links.Delete(linkToUpdate);
285     }
286     else if (substitution.Count == 3)
287     {
288         Links.Update(after);
289     }
290     else
291     {
292         throw new NotSupportedException();
293     }
294     if (matchHandler != null)
295     {
296         return substitutionHandler(before, after);
297     }
298     return Constants.Continue;
299 }
300 else
301 {
302     throw new NotSupportedException();
303 }
304 }
305 }
306
307 /// <remarks>
308 /// IList[IList[IList[T]]]
309 /// |         |         |
310 /// |         |         |-----|
311 /// |         |         |   link   |
312 /// |         |         |-----|
313 /// |         |         |   change  |
314 /// |         |         |-----|
315 /// |         |         |   changes  |
316 /// </remarks>
317 public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition,
318     ↪ IList<TLink> substitution)
319 {
320     var changes = new List<IList<IList<TLink>>>();
321     Trigger(condition, AlwaysContinue, substitution, (before, after) =>
322     {
323         var change = new[] { before, after };
324         changes.Add(change);
325         return Constants.Continue;
326     });
327     return changes;
328 }
329
330 private TLink AlwaysContinue(IList<TLink> linkToMatch) =>
331     ↪ Constants.Continue;
332 }
333 }

```

./DoubletComparer.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Data.Doublets
5  {
6      /// <remarks>
7      /// TODO: Может стоит попробовать ref во всех методах
8      ↪ (IRefEqualityComparer)
9      /// 2x faster with comparer
10     /// </remarks>
11     public class DoubletComparer<T> : IEqualityComparer<Doublet<T>>

```

```

11     {
12         private static readonly EqualityComparer<T> _equalityComparer =
13             ↪ EqualityComparer<T>.Default;
14
15         public static readonly DoubletComparer<T> Default = new
16             ↪ DoubletComparer<T>();
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public bool Equals(Doublet<T> x, Doublet<T> y) =>
20             ↪ _equalityComparer.Equals(x.Source, y.Source) &&
21             ↪ _equalityComparer.Equals(x.Target, y.Target);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public int GetHashCode(Doublet<T> obj) =>
25             ↪ unchecked(obj.Source.GetHashCode() << 15 ^
26             ↪ obj.Target.GetHashCode());
27     }
28 }

```

./Doublet.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets
5  {
6      public struct Doublet<T> : IEquatable<Doublet<T>>
7      {
8          private static readonly EqualityComparer<T> _equalityComparer =
9              ↪ EqualityComparer<T>.Default;
10
11          public T Source { get; set; }
12          public T Target { get; set; }
13
14          public Doublet(T source, T target)
15          {
16              Source = source;
17              Target = target;
18          }
19
20          public override string ToString() => $"{Source}->{Target}";
21
22          public bool Equals(Doublet<T> other) =>
23              ↪ _equalityComparer.Equals(Source, other.Source) &&
24              ↪ _equalityComparer.Equals(Target, other.Target);
25      }
26 }

```

./Hybrid.cs

```

1  using System;
2  using System.Reflection;
3  using Platform.Reflection;
4  using Platform.Converters;
5  using Platform.Numbers;
6
7  namespace Platform.Data.Doublets
8  {
9      public class Hybrid<T>
10     {
11         public readonly T Value;
12         public bool IsNothing => Convert.ToInt64(To.Signed(Value)) == 0;
13         public bool IsInternal => Convert.ToInt64(To.Signed(Value)) > 0;
14         public bool IsExternal => Convert.ToInt64(To.Signed(Value)) < 0;
15         public long AbsoluteValue =>
16             ↪ Math.Abs(Convert.ToInt64(To.Signed(Value)));
17     }
18 }

```

```

17 public Hybrid(T value)
18 {
19     if (CachedTypeInfo<T>.IsSigned)
20     {
21         throw new NotSupportedException();
22     }
23     Value = value;
24 }
25
26 public Hybrid(object value) => Value =
    ↳ To.UnsignedAs<T>(Convert.ChangeType(value,
    ↳ CachedTypeInfo<T>.SignedVersion));
27
28 public Hybrid(object value, bool isExternal)
29 {
30     var signedType = CachedTypeInfo<T>.SignedVersion;
31     var signedValue = Convert.ChangeType(value, signedType);
32     var abs = typeof(MathHelpers).GetTypeInfo().GetMethod("Abs").MakeG
    ↳ enericMethod(signedType);
33     var negate = typeof(MathHelpers).GetTypeInfo().GetMethod("Negate")
    ↳ .MakeGenericMethod(signedType);
34     var absoluteValue = abs.Invoke(null, new[] { signedValue });
35     var resultValue = isExternal ? negate.Invoke(null, new[] {
    ↳ absoluteValue }) : absoluteValue;
36     Value = To.UnsignedAs<T>(resultValue);
37 }
38
39 public static implicit operator Hybrid<T>(T integer) => new
    ↳ Hybrid<T>(integer);
40
41 public static explicit operator Hybrid<T>(ulong integer) => new
    ↳ Hybrid<T>(integer);
42
43 public static explicit operator Hybrid<T>(long integer) => new
    ↳ Hybrid<T>(integer);
44
45 public static explicit operator Hybrid<T>(uint integer) => new
    ↳ Hybrid<T>(integer);
46
47 public static explicit operator Hybrid<T>(int integer) => new
    ↳ Hybrid<T>(integer);
48
49 public static explicit operator Hybrid<T>(ushort integer) => new
    ↳ Hybrid<T>(integer);
50
51 public static explicit operator Hybrid<T>(short integer) => new
    ↳ Hybrid<T>(integer);
52
53 public static explicit operator Hybrid<T>(byte integer) => new
    ↳ Hybrid<T>(integer);
54
55 public static explicit operator Hybrid<T>(sbyte integer) => new
    ↳ Hybrid<T>(integer);
56
57 public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;
58
59 public static explicit operator ulong(Hybrid<T> hybrid) =>
    ↳ Convert.ToUInt64(hybrid.Value);
60
61 public static explicit operator long(Hybrid<T> hybrid) =>
    ↳ hybrid.AbsoluteValue;
62
63 public static explicit operator uint(Hybrid<T> hybrid) =>
    ↳ Convert.ToUInt32(hybrid.Value);
64

```

```

65 public static explicit operator int(Hybrid<T> hybrid) =>
    ↳ Convert.ToInt32(hybrid.AbsoluteValue);
66
67 public static explicit operator ushort(Hybrid<T> hybrid) =>
    ↳ Convert.ToUInt16(hybrid.Value);
68
69 public static explicit operator short(Hybrid<T> hybrid) =>
    ↳ Convert.ToInt16(hybrid.AbsoluteValue);
70
71 public static explicit operator byte(Hybrid<T> hybrid) =>
    ↳ Convert.ToByte(hybrid.Value);
72
73 public static explicit operator sbyte(Hybrid<T> hybrid) =>
    ↳ Convert.ToSByte(hybrid.AbsoluteValue);
74
75 public override string ToString() => IsNothing ? default(T) == null ?
    ↳ "Nothing" : default(T).ToString() : IsExternal ?
    ↳ $"{AbsoluteValue}" : Value.ToString();
76     }
77 }

```

./ILinks.cs

```

1 using Platform.Data.Constants;
2
3 namespace Platform.Data.Doublets
4 {
5     public interface ILinks<TLink> : ILinks<TLink,
    ↳ LinksCombinedConstants<TLink, TLink, int>>
6     {
7     }
8 }

```

./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.Numbers;
9 using Platform.Random;
10 using Platform.Helpers.Setters;
11 using Platform.Data.Exceptions;
12
13 namespace Platform.Data.Doublets
14 {
15     public static class ILinksExtensions
16     {
17         public static void RunRandomCreations<TLink>(this ILinks<TLink> links,
    ↳ long amountOfCreations)
18         {
19             for (long i = 0; i < amountOfCreations; i++)
20             {
21                 var linksAddressRange = new Range<ulong>(0,
    ↳ (Integer<TLink>)links.Count());
22                 Integer<TLink> source =
    ↳ RandomHelpers.Default.NextUInt64(linksAddressRange);
23                 Integer<TLink> target =
    ↳ RandomHelpers.Default.NextUInt64(linksAddressRange);
24                 links.CreateAndUpdate(source, target);
25             }
26         }
27     }

```

```

28 public static void RunRandomSearches<TLink>(this ILinks<TLink> links,
    ↳ long amountOfSearches)
29 {
30     for (long i = 0; i < amountOfSearches; i++)
31     {
32         var linkAddressRange = new Range<ulong>(1,
    ↳ (Integer<TLink>)links.Count());
33         Integer<TLink> source =
    ↳ RandomHelpers.Default.NextUInt64(linkAddressRange);
34         Integer<TLink> target =
    ↳ RandomHelpers.Default.NextUInt64(linkAddressRange);
35         links.SearchOrDefault(source, target);
36     }
37 }
38
39 public static void RunRandomDeletions<TLink>(this ILinks<TLink> links,
    ↳ long amountOfDeletions)
40 {
41     var min = (ulong)amountOfDeletions > (Integer<TLink>)links.Count()
    ↳ ? 1 : (Integer<TLink>)links.Count() - (ulong)amountOfDeletions;
42     for (long i = 0; i < amountOfDeletions; i++)
43     {
44         var linksAddressRange = new Range<ulong>(min,
    ↳ (Integer<TLink>)links.Count());
45         Integer<TLink> link =
    ↳ RandomHelpers.Default.NextUInt64(linksAddressRange);
46         links.Delete(link);
47         if ((Integer<TLink>)links.Count() < min)
48         {
49             break;
50         }
51     }
52 }
53
54 /// <remarks>
55 /// TODO: Возможно есть очень простой способ это сделать.
56 /// (Например просто удалить файл, или изменить его размер таким
    ↳ образом,
57 /// чтобы удалился весь контент)
58 /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
59 /// </remarks>
60 public static void DeleteAll<TLink>(this ILinks<TLink> links)
61 {
62     var equalityComparer = EqualityComparer<TLink>.Default;
63     var comparer = Comparer<TLink>.Default;
64     for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
    ↳ ArithmeticHelpers.Decrement(i))
65     {
66         links.Delete(i);
67         if (!equalityComparer.Equals(links.Count(),
    ↳ ArithmeticHelpers.Decrement(i)))
68         {
69             i = links.Count();
70         }
71     }
72 }
73
74 public static TLink First<TLink>(this ILinks<TLink> links)
75 {
76     TLink firstLink = default;
77     var equalityComparer = EqualityComparer<TLink>.Default;
78     if (equalityComparer.Equals(links.Count(), default))
79     {
80         throw new Exception("В хранилище нет связей.");
81     }

```

```

82 links.Each(links.Constants.Any, links.Constants.Any, link =>
83 {
84     firstLink = link[links.Constants.IndexPart];
85     return links.Constants.Break;
86 });
87 if (equalityComparer.Equals(firstLink, default))
88 {
89     throw new Exception("В процессе поиска по хранилищу не было
    ↳ найдено связей.");
90 }
91 return firstLink;
92 }
93
94 public static bool IsInnerReference<TLink>(this ILinks<TLink> links,
    ↳ TLink reference)
95 {
96     var constants = links.Constants;
97     var comparer = Comparer<TLink>.Default;
98     return comparer.Compare(constants.MinPossibleIndex, reference) >=
    ↳ 0 && comparer.Compare(reference, constants.MaxPossibleIndex)
    ↳ <= 0;
99 }
100
101 #region Paths
102
103 /// <remarks>
104 /// TODO: Как так? Как то что ниже может быть корректно?
105 /// Скорее всего практически не применимо
106 /// Предполагалось, что можно было конвертировать формируемый в
    ↳ проходе через SequenceWalker
107 /// Stack в конкретный путь из Source, Target до связи, но это не
    ↳ всегда так.
108 /// TODO: Возможно нужен метод, который именно выбрасывает исключения
    ↳ (EnsurePathExists)
109 /// </remarks>
110 public static bool CheckPathExistance<TLink>(this ILinks<TLink> links,
    ↳ params TLink[] path)
111 {
112     var current = path[0];
113     //EnsureLinkExists(current, "path");
114     if (!links.Exists(current))
115     {
116         return false;
117     }
118     var equalityComparer = EqualityComparer<TLink>.Default;
119     var constants = links.Constants;
120     for (var i = 1; i < path.Length; i++)
121     {
122         var next = path[i];
123         var values = links.GetLink(current);
124         var source = values[constants.SourcePart];
125         var target = values[constants.TargetPart];
126         if (equalityComparer.Equals(source, target) &&
    ↳ equalityComparer.Equals(source, next))
127         {
128             //throw new Exception(string.Format("Невозможно выбрать
    ↳ путь, так как и Source и Target совпадают с элементом
    ↳ пути {0}.", next));
            return false;
129         }
130     }
131     if (!equalityComparer.Equals(next, source) &&
    ↳ !equalityComparer.Equals(next, target))
132     {
133         //throw new Exception(string.Format("Невозможно продолжить
    ↳ путь через элемент пути {0}", next));

```

```

134         return false;
135     }
136     current = next;
137 }
138 return true;
139 }
140
141 /// <remarks>
142 /// Может потребовать дополнительного стека для PathElement's при
143   ↳ использовании SequenceWalker.
144 /// </remarks>
145 public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink
146   ↳ root, params int[] path)
147 {
148     links.EnsureLinkExists(root, "root");
149     var currentLink = root;
150     for (var i = 0; i < path.Length; i++)
151     {
152         currentLink = links.GetLink(currentLink)[path[i]];
153     }
154     return currentLink;
155 }
156
157 public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this
158   ↳ ILinks<TLink> links, TLink root, ulong size, ulong index)
159 {
160     var constants = links.Constants;
161     var source = constants.SourcePart;
162     var target = constants.TargetPart;
163     if (!MathHelpers.IsPowerOfTwo(size))
164     {
165         throw new ArgumentOutOfRangeException(nameof(size), "Sequences
166           ↳ with sizes other than powers of two are not supported.");
167     }
168     var path = new BitArray(BitConverter.GetBytes(index));
169     var length = BitwiseHelpers.GetLowestBitPosition(size);
170     links.EnsureLinkExists(root, "root");
171     var currentLink = root;
172     for (var i = length - 1; i >= 0; i--)
173     {
174         currentLink = links.GetLink(currentLink)[path[i] ? target :
175           ↳ source];
176     }
177     return currentLink;
178 }
179
180 #endregion
181
182 /// <summary>
183 /// Возвращает индекс указанной связи.
184 /// </summary>
185 /// <param name="links">Хранилище связей.</param>
186 /// <param name="link">Связь представленная списком, состоящим из её
187   ↳ адреса и содержимого.</param>
188 /// <returns>Индекс начальной связи для указанной связи.</returns>
189 [MethodImpl(MethodImplOptions.AggressiveInlining)]
190 public static TLink GetIndex<TLink>(this ILinks<TLink> links,
191   ↳ IList<TLink> link) => link[links.Constants.IndexPart];
192
193 /// <summary>
194 /// Возвращает индекс начальной (Source) связи для указанной связи.
195 /// </summary>
196 /// <param name="links">Хранилище связей.</param>
197 /// <param name="link">Индекс связи.</param>
198 /// <returns>Индекс начальной связи для указанной связи.</returns>
199 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

193 public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink
194   ↳ link) => links.GetLink(link)[links.Constants.SourcePart];
195
196 /// <summary>
197 /// Возвращает индекс начальной (Source) связи для указанной связи.
198 /// </summary>
199 /// <param name="links">Хранилище связей.</param>
200 /// <param name="link">Связь представленная списком, состоящим из её
201   ↳ адреса и содержимого.</param>
202 /// <returns>Индекс начальной связи для указанной связи.</returns>
203 [MethodImpl(MethodImplOptions.AggressiveInlining)]
204 public static TLink GetSource<TLink>(this ILinks<TLink> links,
205   ↳ IList<TLink> link) => link[links.Constants.SourcePart];
206
207 /// <summary>
208 /// Возвращает индекс конечной (Target) связи для указанной связи.
209 /// </summary>
210 /// <param name="links">Хранилище связей.</param>
211 /// <param name="link">Индекс связи.</param>
212 /// <returns>Индекс конечной связи для указанной связи.</returns>
213 [MethodImpl(MethodImplOptions.AggressiveInlining)]
214 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink
215   ↳ link) => links.GetLink(link)[links.Constants.TargetPart];
216
217 /// <summary>
218 /// Возвращает индекс конечной (Target) связи для указанной связи.
219 /// </summary>
220 /// <param name="links">Хранилище связей.</param>
221 /// <param name="link">Связь представленная списком, состоящим из её
222   ↳ адреса и содержимого.</param>
223 /// <returns>Индекс конечной связи для указанной связи.</returns>
224 [MethodImpl(MethodImplOptions.AggressiveInlining)]
225 public static TLink GetTarget<TLink>(this ILinks<TLink> links,
226   ↳ IList<TLink> link) => link[links.Constants.TargetPart];
227
228 /// <summary>
229 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая
230   ↳ обработчик (handler) для каждой подходящей связи.
231 /// </summary>
232 /// <param name="links">Хранилище связей.</param>
233 /// <param name="handler">Обработчик каждой подходящей связи.</param>
234 /// <param name="restrictions">Ограничения на содержимое связей.
235   ↳ Каждое ограничение может иметь значения: Constants.Null - 0-я
236   ↳ связь, обозначающая ссылку на пустоту, Any - отсутствие
237   ↳ ограничения, 1..∞ конкретный адрес связи.</param>
238 /// <returns>True, в случае если проход по связям не был прерван и
239   ↳ False в обратном случае.</returns>
240 [MethodImpl(MethodImplOptions.AggressiveInlining)]
241 public static bool Each<TLink>(this ILinks<TLink> links,
242   ↳ Func<IList<TLink>, TLink> handler, params TLink[] restrictions)
243   => EqualityComparer<TLink>.Default.Equals(links.Each(handler,
244     ↳ restrictions), links.Constants.Continue);
245
246 /// <summary>
247 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая
248   ↳ обработчик (handler) для каждой подходящей связи.
249 /// </summary>
250 /// <param name="links">Хранилище связей.</param>
251 /// <param name="source">Значение, определяющее соответствующие
252   ↳ шаблону связи. (Constants.Null - 0-я связь, обозначающая ссылку на
253   ↳ пустоту в качестве начала, Constants.Any - любое начало, 1..∞
254   ↳ конкретное начало)</param>

```

```

238 /// <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>
239 /// <param name="handler">Обработчик каждой подходящей связи.</param>
240 /// <returns>True, в случае если проход по связям не был прерван и
    ↳ False в обратном случае.</returns>
241 [MethodImpl(MethodImplOptions.AggressiveInlining)]
242 public static bool Each<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target, Func<TLink, bool> handler)
243 {
244     var constants = links.Constants;
245     return links.Each(link => handler(link[constants.IndexPart]) ?
    ↳ constants.Continue : constants.Break, constants.Any, source,
    ↳ target);
246 }
247
248 /// <summary>
249 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая
    ↳ обработчик (handler) для каждой подходящей связи.
250 /// </summary>
251 /// <param name="links">Хранилище связей.</param>
252 /// <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>
253 /// <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>
254 /// <param name="handler">Обработчик каждой подходящей связи.</param>
255 /// <returns>True, в случае если проход по связям не был прерван и
    ↳ False в обратном случае.</returns>
256 [MethodImpl(MethodImplOptions.AggressiveInlining)]
257 public static bool Each<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target, Func<IList<TLink>, TLink> handler)
258 {
259     var constants = links.Constants;
260     return links.Each(handler, constants.Any, source, target);
261 }
262
263 [MethodImpl(MethodImplOptions.AggressiveInlining)]
264 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links,
    ↳ params TLink[] restrictions)
265 {
266     var constants = links.Constants;
267     int listSize = (Integer<TLink>)links.Count(restrictions);
268     var list = new IList<TLink>[listSize];
269     if (listSize > 0)
270     {
271         var filler = new ArrayFiller<IList<TLink>, TLink>(list,
    ↳ links.Constants.Continue);
272         links.Each(filler.AddAndReturnConstant, restrictions);
273     }
274     return list;
275 }
276
277 /// <summary>
278 /// Возвращает значение, определяющее существует ли связь с указанными
    ↳ началом и концом в хранилище связей.
279 /// </summary>
280 /// <param name="links">Хранилище связей.</param>
281 /// <param name="source">Начало связи.</param>
282 /// <param name="target">Конец связи.</param>

```

```

283 /// <returns>Значение, определяющее существует ли связь.</returns>
284 [MethodImpl(MethodImplOptions.AggressiveInlining)]
285 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;
286
287 #region Ensure
288 // TODO: May be move to EnsureExtensions or make it both there and here
289
290 [MethodImpl(MethodImplOptions.AggressiveInlining)]
291 public static void EnsureInnerReferenceExists<TLink>(this
    ↳ ILinks<TLink> links, TLink reference, string argumentName)
292 {
293     if (links.IsInnerReference(reference) && !links.Exists(reference))
294     {
295         throw new ArgumentLinkDoesNotExistsException<TLink>(reference,
    ↳ argumentName);
296     }
297 }
298
299 [MethodImpl(MethodImplOptions.AggressiveInlining)]
300 public static void EnsureInnerReferenceExists<TLink>(this
    ↳ ILinks<TLink> links, IList<TLink> restrictions, string
    ↳ argumentName)
301 {
302     for (int i = 0; i < restrictions.Count; i++)
303     {
304         links.EnsureInnerReferenceExists(restrictions[i],
    ↳ argumentName);
305     }
306 }
307
308 [MethodImpl(MethodImplOptions.AggressiveInlining)]
309 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink>
    ↳ links, IList<TLink> restrictions)
310 {
311     for (int i = 0; i < restrictions.Count; i++)
312     {
313         links.EnsureLinkIsAnyOrExists(restrictions[i],
    ↳ nameof(restrictions));
314     }
315 }
316
317 [MethodImpl(MethodImplOptions.AggressiveInlining)]
318 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink>
    ↳ links, TLink link, string argumentName)
319 {
320     var equalityComparer = EqualityComparer<TLink>.Default;
321     if (!equalityComparer.Equals(link, links.Constants.Any) &&
    ↳ !links.Exists(link))
322     {
323         throw new ArgumentLinkDoesNotExistsException<TLink>(link,
    ↳ argumentName);
324     }
325 }
326
327 [MethodImpl(MethodImplOptions.AggressiveInlining)]
328 public static void EnsureLinkIsItselfOrExists<TLink>(this
    ↳ ILinks<TLink> links, TLink link, string argumentName)
329 {
330     var equalityComparer = EqualityComparer<TLink>.Default;
331     if (!equalityComparer.Equals(link, links.Constants.Itself) &&
    ↳ !links.Exists(link))

```

```

332     {
333         throw new ArgumentLinkDoesNotExistsException<TLink>(link,
            ↳ argumentName);
334     }
335 }
336
337 /// <param name="links">Хранилище связей.</param>
338 [MethodImpl(MethodImplOptions.AggressiveInlining)]
339 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink>
    ↳ links, TLink source, TLink target)
340 {
341     if (links.Exists(source, target))
342     {
343         throw new LinkWithSameValueAlreadyExistsException();
344     }
345 }
346
347 /// <param name="links">Хранилище связей.</param>
348 public static void EnsureNoDependencies<TLink>(this ILinks<TLink>
    ↳ links, TLink link)
349 {
350     if (links.DependenciesExist(link))
351     {
352         throw new ArgumentLinkHasDependenciesException<TLink>(link);
353     }
354 }
355
356 /// <param name="links">Хранилище связей.</param>
357 public static void EnsureCreated<TLink>(this ILinks<TLink> links,
    ↳ params TLink[] addresses) => links.EnsureCreated(links.Create,
    ↳ addresses);
358
359 /// <param name="links">Хранилище связей.</param>
360 public static void EnsurePointsCreated<TLink>(this ILinks<TLink>
    ↳ links, params TLink[] addresses) =>
    ↳ links.EnsureCreated(links.CreatePoint, addresses);
361
362 /// <param name="links">Хранилище связей.</param>
363 public static void EnsureCreated<TLink>(this ILinks<TLink> links,
    ↳ Func<TLink> creator, params TLink[] addresses)
364 {
365     var constants = links.Constants;
366     var nonExistentAddresses = new HashSet<ulong>(addresses.Where(x =>
    ↳ !links.Exists(x).Select(x => (ulong)(Integer<TLink>)x)));
367     if (nonExistentAddresses.Count > 0)
368     {
369         var max = nonExistentAddresses.Max();
370         // TODO: Эту верхнюю границу нужно разрешить переопределять
371         ↳ (проверить применяется ли эта логика)
372         max = Math.Min(max,
373             ↳ (Integer<TLink>)constants.MaxPossibleIndex);
374         var createdLinks = new List<TLink>();
375         var equalityComparer = EqualityComparer<TLink>.Default;
376         TLink createdLink = creator();
377         while (!equalityComparer.Equals(createdLink,
378             ↳ (Integer<TLink>)max))
379         {
380             createdLinks.Add(createdLink);
381         }
382         for (var i = 0; i < createdLinks.Count; i++)
383         {
384             if (!nonExistentAddresses.Contains((Integer<TLink>)created_
385                 ↳ Links[i]))
386             {

```

```

383         links.Delete(createdLinks[i]);
384     }
385 }
386 }
387 }
388 }
389 #endregion
390
391 /// <param name="links">Хранилище связей.</param>
392 public static ulong DependenciesCount<TLink>(this ILinks<TLink> links,
    ↳ TLink link)
393 {
394     var constants = links.Constants;
395     var values = links.GetLink(link);
396     ulong referencesAsSource =
397         ↳ (Integer<TLink>)links.Count(constants.Any, link,
398         ↳ constants.Any);
399     var equalityComparer = EqualityComparer<TLink>.Default;
400     if (equalityComparer.Equals(values[constants.SourcePart], link))
401     {
402         referencesAsSource--;
403     }
404     ulong referencesAsTarget =
405         ↳ (Integer<TLink>)links.Count(constants.Any, constants.Any,
406         ↳ link);
407     if (equalityComparer.Equals(values[constants.TargetPart], link))
408     {
409         referencesAsTarget--;
410     }
411     return referencesAsSource + referencesAsTarget;
412 }
413
414 /// <param name="links">Хранилище связей.</param>
415 [MethodImpl(MethodImplOptions.AggressiveInlining)]
416 public static bool DependenciesExist<TLink>(this ILinks<TLink> links,
    ↳ TLink link) => links.DependenciesCount(link) > 0;
417
418 /// <param name="links">Хранилище связей.</param>
419 [MethodImpl(MethodImplOptions.AggressiveInlining)]
420 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link,
    ↳ TLink source, TLink target)
421 {
422     var constants = links.Constants;
423     var values = links.GetLink(link);
424     var equalityComparer = EqualityComparer<TLink>.Default;
425     return equalityComparer.Equals(values[constants.SourcePart],
426         ↳ source) &&
427         equalityComparer.Equals(values[constants.TargetPart], target);
428 }
429
430 /// <summary>
431 /// Выполняет поиск связи с указанными Source (началом) и Target
432 /// (концом).
433 /// </summary>
434 /// <param name="links">Хранилище связей.</param>
435 /// <param name="source">Индекс связи, которая является началом для
436   ↳ искомой связи.</param>
437 /// <param name="target">Индекс связи, которая является концом для
438   ↳ искомой связи.</param>
439 /// <returns>Индекс искомой связи с указанными Source (началом) и
440   ↳ Target (концом).</returns>
441 [MethodImpl(MethodImplOptions.AggressiveInlining)]
442 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links,
    ↳ TLink source, TLink target)
443 {

```



```

434     var constants = links.Constants;
435     var setter = new Setter<TLink, TLink>(constants.Continue,
        ↳ constants.Break, default);
436     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source,
        ↳ target);
437     return setter.Result;
438 }
439
440 /// <param name="links">Хранилище связей.</param>
441 [MethodImpl(MethodImplOptions.AggressiveInlining)]
442 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
443 {
444     var link = links.Create();
445     return links.Update(link, link, link);
446 }
447
448 /// <param name="links">Хранилище связей.</param>
449 [MethodImpl(MethodImplOptions.AggressiveInlining)]
450 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links,
    ↳ TLink source, TLink target) => links.Update(links.Create(),
    ↳ source, target);
451
452 /// <summary>
453 /// Обновляет связь с указанными началом (Source) и концом (Target)
454 /// на связь с указанными началом (NewSource) и концом (NewTarget).
455 /// </summary>
456 /// <param name="links">Хранилище связей.</param>
457 /// <param name="link">Индекс обновляемой связи.</param>
458 /// <param name="newSource">Индекс связи, которая является началом
    ↳ связи, на которую выполняется обновление.</param>
459 /// <param name="newTarget">Индекс связи, которая является концом
    ↳ связи, на которую выполняется обновление.</param>
460 /// <returns>Индекс обновлённой связи.</returns>
461 [MethodImpl(MethodImplOptions.AggressiveInlining)]
462 public static TLink Update<TLink>(this ILinks<TLink> links, TLink
    ↳ link, TLink newSource, TLink newTarget) => links.Update(new[] {
    ↳ link, newSource, newTarget });
463
464 /// <summary>
465 /// Обновляет связь с указанными началом (Source) и концом (Target)
466 /// на связь с указанными началом (NewSource) и концом (NewTarget).
467 /// </summary>
468 /// <param name="links">Хранилище связей.</param>
469 /// <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>
470 /// <returns>Индекс обновлённой связи.</returns>
471 [MethodImpl(MethodImplOptions.AggressiveInlining)]
472 public static TLink Update<TLink>(this ILinks<TLink> links, params
    ↳ TLink[] restrictions)
473 {
474     if (restrictions.Length == 2)
475     {
476         return links.Merge(restrictions[0], restrictions[1]);
477     }
478     if (restrictions.Length == 4)
479     {
480         return links.UpdateOrCreateOrGet(restrictions[0],
            ↳ restrictions[1], restrictions[2], restrictions[3]);
481     }
482     else
483     {
484         return links.Update(restrictions);

```

```

485     }
486 }
487
488 /// <summary>
489 /// Создаёт связь (если она не существовала), либо возвращает индекс
    ↳ существующей связи с указанными Source (началом) и Target (концом).
490 /// </summary>
491 /// <param name="links">Хранилище связей.</param>
492 /// <param name="source">Индекс связи, которая является началом на
    ↳ создаваемой связи.</param>
493 /// <param name="target">Индекс связи, которая является концом для
    ↳ создаваемой связи.</param>
494 /// <returns>Индекс связи, с указанным Source (началом) и Target
    ↳ (концом)</returns>
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink
    ↳ source, TLink target)
497 {
498     var link = links.SearchOrDefault(source, target);
499     if (EqualityComparer<TLink>.Default.Equals(link, default))
500     {
501         link = links.CreateAndUpdate(source, target);
502     }
503     return link;
504 }
505
506 /// <summary>
507 /// Обновляет связь с указанными началом (Source) и концом (Target)
508 /// на связь с указанными началом (NewSource) и концом (NewTarget).
509 /// </summary>
510 /// <param name="links">Хранилище связей.</param>
511 /// <param name="source">Индекс связи, которая является началом
    ↳ обновляемой связи.</param>
512 /// <param name="target">Индекс связи, которая является концом
    ↳ обновляемой связи.</param>
513 /// <param name="newSource">Индекс связи, которая является началом
    ↳ связи, на которую выполняется обновление.</param>
514 /// <param name="newTarget">Индекс связи, которая является концом
    ↳ связи, на которую выполняется обновление.</param>
515 /// <returns>Индекс обновлённой связи.</returns>
516 [MethodImpl(MethodImplOptions.AggressiveInlining)]
517 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink>
    ↳ links, TLink source, TLink target, TLink newSource, TLink
    ↳ newTarget)
518 {
519     var equalityComparer = EqualityComparer<TLink>.Default;
520     var link = links.SearchOrDefault(source, target);
521     if (equalityComparer.Equals(link, default))
522     {
523         return links.CreateAndUpdate(newSource, newTarget);
524     }
525     if (equalityComparer.Equals(newSource, source) &&
        ↳ equalityComparer.Equals(newTarget, target))
526     {
527         return link;
528     }
529     return links.Update(link, newSource, newTarget);
530 }
531
532 /// <summary>Удаляет связь с указанными началом (Source) и концом
    ↳ (Target).</summary>
533 /// <param name="links">Хранилище связей.</param>
534 /// <param name="source">Индекс связи, которая является началом
    ↳ удаляемой связи.</param>

```



```

535 /// <param name="target">Индекс связи, которая является концом
    ↳ удаляемой связи.</param>
536 [MethodImpl(MethodImplOptions.AggressiveInlining)]
537 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links,
    ↳ TLink source, TLink target)
538 {
539     var link = links.SearchOrDefault(source, target);
540     if (!EqualityComparer<TLink>.Default.Equals(link, default))
541     {
542         links.Delete(link);
543         return link;
544     }
545     return default;
546 }
547
548 /// <summary>Удаляет несколько связей.</summary>
549 /// <param name="links">Хранилище связей.</param>
550 /// <param name="deletedLinks">Список адресов связей к
    ↳ удалению.</param>
551 [MethodImpl(MethodImplOptions.AggressiveInlining)]
552 public static void DeleteMany<TLink>(this ILinks<TLink> links,
    ↳ IList<TLink> deletedLinks)
553 {
554     for (int i = 0; i < deletedLinks.Count; i++)
555     {
556         links.Delete(deletedLinks[i]);
557     }
558 }
559
560 // Replace one link with another (replaced link is deleted, children
    ↳ are updated or deleted)
561 public static TLink Merge<TLink>(this ILinks<TLink> links, TLink
    ↳ linkIndex, TLink newLink)
562 {
563     var equalityComparer = EqualityComparer<TLink>.Default;
564     if (equalityComparer.Equals(linkIndex, newLink))
565     {
566         return newLink;
567     }
568     var constants = links.Constants;
569     ulong referencesAsSourceCount =
    ↳ (Integer<TLink>)links.Count(constants.Any, linkIndex,
    ↳ constants.Any);
570     ulong referencesAsTargetCount =
    ↳ (Integer<TLink>)links.Count(constants.Any, constants.Any,
    ↳ linkIndex);
571     var isStandalonePoint =
    ↳ Point<TLink>.IsFullPoint(links.GetLink(linkIndex)) &&
    ↳ referencesAsSourceCount == 1 && referencesAsTargetCount == 1;
572     if (!isStandalonePoint)
573     {
574         var totalReferences = referencesAsSourceCount +
    ↳ referencesAsTargetCount;
575         if (totalReferences > 0)
576         {
577             var references =
    ↳ ArrayPool.Allocate<TLink>((long)totalReferences);
578             var referencesFiller = new ArrayFiller<TLink,
    ↳ TLink>(references, links.Constants.Continue);
579             links.Each(referencesFiller.AddFirstAndReturnConstant,
    ↳ constants.Any, linkIndex, constants.Any);
580             links.Each(referencesFiller.AddFirstAndReturnConstant,
    ↳ constants.Any, constants.Any, linkIndex);
581             for (ulong i = 0; i < referencesAsSourceCount; i++)
582             {

```

```

583         var reference = references[i];
584         if (equalityComparer.Equals(reference, linkIndex))
585         {
586             continue;
587         }
588
589         links.Update(reference, newLink,
    ↳ links.GetTarget(reference));
590     }
591     for (var i = (long)referencesAsSourceCount; i <
    ↳ references.Length; i++)
592     {
593         var reference = references[i];
594         if (equalityComparer.Equals(reference, linkIndex))
595         {
596             continue;
597         }
598
599         links.Update(reference, links.GetSource(reference),
    ↳ newLink);
600     }
601     ArrayPool.Free(references);
602 }
603 }
604 links.Delete(linkIndex);
605 return newLink;
606 }
607 }
608 }

```

./Incrementers/FrequencyIncrementer.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Incrementers
5 {
6     public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>,
    ↳ IIncrementer<TLink>
7     {
8         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↳ EqualityComparer<TLink>.Default;
9
10         private readonly TLink _frequencyMarker;
11         private readonly TLink _unaryOne;
12         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
13
14         public FrequencyIncrementer(ILinks<TLink> links, TLink
    ↳ frequencyMarker, TLink unaryOne, IIncrementer<TLink>
    ↳ unaryNumberIncrementer)
    ↳ : base(links)
15         {
16             _frequencyMarker = frequencyMarker;
17             _unaryOne = unaryOne;
18             _unaryNumberIncrementer = unaryNumberIncrementer;
19         }
20
21         public TLink Increment(TLink frequency)
22         {
23             if (_equalityComparer.Equals(frequency, default))
24             {
25                 return Links.GetOrCreate(_unaryOne, _frequencyMarker);
26             }
27             var source = Links.GetSource(frequency);
28             var incrementedSource = _unaryNumberIncrementer.Increment(source);
29             return Links.GetOrCreate(incrementedSource, _frequencyMarker);
30         }
31     }

```

```

32     }
33 }

./Incrementers/LinkFrequencyIncrementer.cs
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Incrementers
5  {
6      public class LinkFrequencyIncrementer<TLink> : LinksOperatorBase<TLink>,
7          ↳ IIncrementer<IList<TLink>>
8      {
9          private readonly ISpecificPropertyOperator<TLink, TLink>
10             ↳ _frequencyPropertyOperator;
11         private readonly IIncrementer<TLink> _frequencyIncrementer;
12
13         public LinkFrequencyIncrementer(ILinks<TLink> links,
14             ↳ ISpecificPropertyOperator<TLink, TLink> frequencyPropertyOperator,
15             ↳ IIncrementer<TLink> frequencyIncrementer)
16             : base(links)
17         {
18             _frequencyPropertyOperator = frequencyPropertyOperator;
19             _frequencyIncrementer = frequencyIncrementer;
20
21             /// <remarks>Sequence itseft is not changed, only frequency of its
22             ↳ doublets is incremented.</remarks>
23             public IList<TLink> Increment(IList<TLink> sequence) // TODO: May be
24                 ↳ move to ILinksExtensions or make
25                 ↳ SequenceDoubletsFrequencyIncrementer
26             {
27                 for (var i = 1; i < sequence.Count; i++)
28                 {
29                     Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
30                 }
31                 return sequence;
32             }
33
34             public void Increment(TLink link)
35             {
36                 var previousFrequency = _frequencyPropertyOperator.Get(link);
37                 var frequency = _frequencyIncrementer.Increment(previousFrequency);
38                 _frequencyPropertyOperator.Set(link, frequency);
39             }
40         }
41     }
42 }

```

```

./Incrementers/UnaryNumberIncrementer.cs
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Incrementers
5  {
6      public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>,
7          ↳ IIncrementer<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↳ EqualityComparer<TLink>.Default;
11
12         private readonly TLink _unaryOne;
13
14         public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) :
15             ↳ base(links) => _unaryOne = unaryOne;
16
17         public TLink Increment(TLink unaryNumber)
18         {
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

```

16         if (_equalityComparer.Equals(unaryNumber, _unaryOne))
17         {
18             return Links.GetOrCreate(_unaryOne, _unaryOne);
19         }
20         var source = Links.GetSource(unaryNumber);
21         var target = Links.GetTarget(unaryNumber);
22         if (_equalityComparer.Equals(source, target))
23         {
24             return Links.GetOrCreate(unaryNumber, _unaryOne);
25         }
26         else
27         {
28             return Links.GetOrCreate(source, Increment(target));
29         }
30     }
31 }
32 }

```

```

./ISynchronizedLinks.cs
1  using Platform.Data.Constants;
2
3  namespace Platform.Data.Doublets
4  {
5      public interface ISynchronizedLinks<TLink> : ISynchronizedLinks<TLink,
6          ↳ ILinks<TLink>, LinksCombinedConstants<TLink, TLink, int>>,
7          ↳ ILinks<TLink>
8      {
9      }
10 }

```

```

./Link.cs
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Platform.Exceptions;
5  using Platform.Ranges;
6  using Platform.Helpers.Singletons;
7  using Platform.Data.Constants;
8
9  namespace Platform.Data.Doublets
10 {
11     /// <summary>
12     /// Структура описывающая уникальную связь.
13     /// </summary>
14     public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>,
15         ↳ IList<TLink>
16     {
17         public static readonly Link<TLink> Null = new Link<TLink>();
18
19         private static readonly LinksCombinedConstants<bool, TLink, int>
20             ↳ _constants = Default<LinksCombinedConstants<bool, TLink,
21             ↳ int>>.Instance;
22         private static readonly EqualityComparer<TLink> _equalityComparer =
23             ↳ EqualityComparer<TLink>.Default;
24
25         private const int Length = 3;
26
27         public readonly TLink Index;
28         public readonly TLink Source;
29         public readonly TLink Target;
30
31         public Link(params TLink[] values)
32         {
33             Index = values.Length > _constants.IndexPart ?
34                 ↳ values[_constants.IndexPart] : _constants.Null;
35             Source = values.Length > _constants.SourcePart ?
36                 ↳ values[_constants.SourcePart] : _constants.Null;
37         }
38     }
39 }

```

```

31     Target = values.Length > _constants.TargetPart ?
    ↪ values[_constants.TargetPart] : _constants.Null;
32 }
33
34 public Link(IList<TLink> values)
35 {
36     Index = values.Count > _constants.IndexPart ?
    ↪ values[_constants.IndexPart] : _constants.Null;
37     Source = values.Count > _constants.SourcePart ?
    ↪ values[_constants.SourcePart] : _constants.Null;
38     Target = values.Count > _constants.TargetPart ?
    ↪ values[_constants.TargetPart] : _constants.Null;
39 }
40
41 public Link(TLink index, TLink source, TLink target)
42 {
43     Index = index;
44     Source = source;
45     Target = target;
46 }
47
48 public Link(TLink source, TLink target)
49 : this(_constants.Null, source, target)
50 {
51     Source = source;
52     Target = target;
53 }
54
55 public static Link<TLink> Create(TLink source, TLink target) => new
    ↪ Link<TLink>(source, target);
56
57 public override int GetHashCode() => (Index, Source,
    ↪ Target).GetHashCode();
58
59 public bool IsNull() => _equalityComparer.Equals(Index,
    ↪ _constants.Null)
60     && _equalityComparer.Equals(Source,
    ↪ _constants.Null)
61     && _equalityComparer.Equals(Target,
    ↪ _constants.Null);
62
63 public override bool Equals(object other) => other is Link<TLink> &&
    ↪ Equals((Link<TLink>)other);
64
65 public bool Equals(Link<TLink> other) =>
    ↪ _equalityComparer.Equals(Index, other.Index)
66     && _equalityComparer.Equals(Sour
    ↪ ce,
    ↪ other.Source)
67     && _equalityComparer.Equals(Targ
    ↪ et,
    ↪ other.Target);
68
69 public static string ToString(TLink index, TLink source, TLink target)
    ↪ => $"({index}: {source}->{target})";
70
71 public static string ToString(TLink source, TLink target) =>
    ↪ $"({source}->{target})";
72
73 public static implicit operator TLink[](Link<TLink> link) =>
    ↪ link.ToArray();
74
75 public static implicit operator Link<TLink>(TLink[] linkArray) => new
    ↪ Link<TLink>(linkArray);
76

```

```

77 public TLink[] ToArray()
78 {
79     var array = new TLink[Length];
80     CopyTo(array, 0);
81     return array;
82 }
83
84 public override string ToString() => _equalityComparer.Equals(Index,
    ↪ _constants.Null) ? ToString(Source, Target) : ToString(Index,
    ↪ Source, Target);
85
86 #region IList
87
88 public int Count => Length;
89
90 public bool IsReadOnly => true;
91
92 public TLink this[int index]
93 {
94     get
95     {
96         Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length
    ↪ - 1), nameof(index));
97         if (index == _constants.IndexPart)
98         {
99             return Index;
100         }
101         if (index == _constants.SourcePart)
102         {
103             return Source;
104         }
105         if (index == _constants.TargetPart)
106         {
107             return Target;
108         }
109         throw new NotSupportedException(); // Impossible path due to
    ↪ Ensure.ArgumentInRange
110     }
111     set => throw new NotSupportedException();
112 }
113
114 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
115
116 public IEnumerator<TLink> GetEnumerator()
117 {
118     yield return Index;
119     yield return Source;
120     yield return Target;
121 }
122
123 public void Add(TLink item) => throw new NotSupportedException();
124
125 public void Clear() => throw new NotSupportedException();
126
127 public bool Contains(TLink item) => IndexOf(item) >= 0;
128
129 public void CopyTo(TLink[] array, int arrayIndex)
130 {
131     Ensure.Always.ArgumentNotNull(array, nameof(array));
132     Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0,
    ↪ array.Length - 1), nameof(arrayIndex));
133     if (arrayIndex + Length > array.Length)
134     {
135         throw new InvalidOperationException();
136     }
137     array[arrayIndex++] = Index;
138     array[arrayIndex++] = Source;

```

```

139     array[arrayIndex] = Target;
140 }
141
142 public bool Remove(TLink item) =>
    ↳ Throw.A.NotSupportedExceptionAndReturn<bool>();
143
144 public int IndexOf(TLink item)
145 {
146     if (_equalityComparer.Equals(Index, item))
147     {
148         return _constants.IndexPart;
149     }
150     if (_equalityComparer.Equals(Source, item))
151     {
152         return _constants.SourcePart;
153     }
154     if (_equalityComparer.Equals(Target, item))
155     {
156         return _constants.TargetPart;
157     }
158     return -1;
159 }
160
161 public void Insert(int index, TLink item) => throw new
    ↳ NotSupportedException();
162
163 public void RemoveAt(int index) => throw new NotSupportedException();
164
165 #endregion
166 }
167 }

```

./LinkExtensions.cs

```

1 namespace Platform.Data.Doublets
2 {
3     public static class LinkExtensions
4     {
5         public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
            ↳ Point<TLink>.IsFullPoint(link);
6         public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
            ↳ Point<TLink>.IsPartialPoint(link);
7     }
8 }

```

./LinksOperatorBase.cs

```

1 namespace Platform.Data.Doublets
2 {
3     public abstract class LinksOperatorBase<TLink>
4     {
5         protected readonly ILinks<TLink> Links;
6         protected LinksOperatorBase(ILinks<TLink> links) => Links = links;
7     }
8 }

```

./obj/Debug/netstandard2.0/Platform.Data.Doublets.AssemblyInfo.cs

```

1 //-----
2 ↳ --
3 // <auto-generated>
4 //     Generated by the MSBuild WriteCodeFragment class.
5 // </auto-generated>
6 ↳ --
7 using System;
8 using System.Reflection;

```

```

9
10 [assembly: System.Reflection.AssemblyConfigurationAttribute("Debug")]
11 [assembly: System.Reflection.AssemblyCopyrightAttribute("Konstantin
    ↳ Diachenko")]
12 [assembly: System.Reflection.AssemblyDescriptionAttribute("LinksPlatform\'s
    ↳ Platform.Data.Doublets Class Library")]
13 [assembly: System.Reflection.AssemblyFileVersionAttribute("0.0.1.0")]
14 [assembly: System.Reflection.AssemblyInformationalVersionAttribute("0.0.1")]
15 [assembly: System.Reflection.AssemblyTitleAttribute("Platform.Data.Doublets")]
16 [assembly: System.Reflection.AssemblyVersionAttribute("0.0.1.0")]

```

./PropertyOperators/DefaultLinkPropertyOperator.cs

```

1 using System.Linq;
2 using System.Collections.Generic;
3 using Platform.Interfaces;
4
5 namespace Platform.Data.Doublets.PropertyOperators
6 {
7     public class DefaultLinkPropertyOperator<TLink> :
            ↳ LinksOperatorBase<TLink>, IPropertyOperator<TLink, TLink, TLink>
8     {
9         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
10
11         public DefaultLinkPropertyOperator(ILinks<TLink> links) : base(links)
12         {
13         }
14
15         public TLink GetValue(TLink @object, TLink property)
16         {
17             var objectProperty = Links.SearchOrDefault(@object, property);
18             if (_equalityComparer.Equals(objectProperty, default))
19             {
20                 return default;
21             }
22             var valueLink = Links.All(Links.Constants.Any,
                ↳ objectProperty).SingleOrDefault();
23             if (valueLink == null)
24             {
25                 return default;
26             }
27             var value = Links.GetTarget(valueLink[Links.Constants.IndexPart]);
28             return value;
29         }
30
31         public void SetValue(TLink @object, TLink property, TLink value)
32         {
33             var objectProperty = Links.GetOrCreate(@object, property);
34             Links.DeleteMany(Links.All(Links.Constants.Any,
                ↳ objectProperty).Select(link =>
                ↳ link[Links.Constants.IndexPart]).ToList());
35             Links.GetOrCreate(objectProperty, value);
36         }
37     }
38 }

```

./PropertyOperators/FrequencyPropertyOperator.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.PropertyOperators
5 {
6     public class FrequencyPropertyOperator<TLink> : LinksOperatorBase<TLink>,
            ↳ ISpecificPropertyOperator<TLink, TLink>
7     {

```

```

8     private static readonly EqualityComparer<TLink> _equalityComparer =
9         ↳ EqualityComparer<TLink>.Default;
10
11     private readonly TLink _frequencyPropertyMarker;
12     private readonly TLink _frequencyMarker;
13
14     public FrequencyPropertyOperator(ILinks<TLink> links, TLink
15         ↳ frequencyPropertyMarker, TLink frequencyMarker) : base(links)
16     {
17         _frequencyPropertyMarker = frequencyPropertyMarker;
18         _frequencyMarker = frequencyMarker;
19     }
20
21     public TLink Get(TLink link)
22     {
23         var property = Links.SearchOrDefault(link,
24             ↳ _frequencyPropertyMarker);
25         var container = GetContainer(property);
26         var frequency = GetFrequency(container);
27         return frequency;
28     }
29
30     private TLink GetContainer(TLink property)
31     {
32         var frequencyContainer = default(TLink);
33         if (_equalityComparer.Equals(property, default))
34         {
35             return frequencyContainer;
36         }
37         Links.Each(candidate =>
38         {
39             var candidateTarget = Links.GetTarget(candidate);
40             var frequencyTarget = Links.GetTarget(candidateTarget);
41             if (_equalityComparer.Equals(frequencyTarget,
42                 ↳ _frequencyMarker))
43             {
44                 frequencyContainer = Links.GetIndex(candidate);
45                 return Links.Constants.Break;
46             }
47             return Links.Constants.Continue;
48         }, Links.Constants.Any, property, Links.Constants.Any);
49         return frequencyContainer;
50     }
51
52     private TLink GetFrequency(TLink container) =>
53         ↳ _equalityComparer.Equals(container, default) ? default :
54         ↳ Links.GetTarget(container);
55
56     public void Set(TLink link, TLink frequency)
57     {
58         var property = Links.GetOrCreate(link, _frequencyPropertyMarker);
59         var container = GetContainer(property);
60         if (_equalityComparer.Equals(container, default))
61         {
62             Links.GetOrCreate(property, frequency);
63         }
64         else
65         {
66             Links.Update(container, property, frequency);
67         }
68     }
69 }

```

./ResizableDirectMemory/ResizableDirectMemoryLinks.cs

```

1     using System;
2     using System.Collections.Generic;
3     using System.Runtime.CompilerServices;
4     using System.Runtime.InteropServices;
5     using Platform.Disposables;
6     using Platform.Helpers.Singletons;
7     using Platform.Collections.Arrays;
8     using Platform.Numbers;
9     using Platform.Unsafe;
10    using Platform.Memory;
11    using Platform.Data.Exceptions;
12    using Platform.Data.Constants;
13    using static Platform.Numbers.ArithmeticHelpers;
14
15    #pragma warning disable 0649
16    #pragma warning disable 169
17    #pragma warning disable 618
18
19    // ReSharper disable StaticMemberInGenericType
20    // ReSharper disable BuiltInTypeReferenceStyle
21    // ReSharper disable MemberCanBePrivate.Local
22    // ReSharper disable UnusedMember.Local
23
24    namespace Platform.Data.Doublets.ResizableDirectMemory
25    {
26        public partial class ResizableDirectMemoryLinks<TLink> : DisposableBase,
27            ↳ ILinks<TLink>
28        {
29            private static readonly EqualityComparer<TLink> _equalityComparer =
30                ↳ EqualityComparer<TLink>.Default;
31            private static readonly Comparer<TLink> _comparer =
32                ↳ Comparer<TLink>.Default;
33
34            /// <summary>Возвращает размер одной связи в байтах.</summary>
35            public static readonly int LinkSizeInBytes =
36                ↳ StructureHelpers.SizeOf<Link>();
37
38            public static readonly int LinkHeaderSizeInBytes =
39                ↳ StructureHelpers.SizeOf<LinksHeader>();
40
41            public static readonly long DefaultLinksSizeStep = LinkSizeInBytes *
42                ↳ 1024 * 1024;
43
44            private struct Link
45            {
46                public static readonly int SourceOffset =
47                    ↳ Marshal.OffsetOf(typeof(Link), nameof(Source)).ToInt32();
48                public static readonly int TargetOffset =
49                    ↳ Marshal.OffsetOf(typeof(Link), nameof(Target)).ToInt32();
50                public static readonly int LeftAsSourceOffset =
51                    ↳ Marshal.OffsetOf(typeof(Link), nameof(LeftAsSource)).ToInt32();
52                public static readonly int RightAsSourceOffset =
53                    ↳ Marshal.OffsetOf(typeof(Link),
54                        ↳ nameof(RightAsSource)).ToInt32();
55                public static readonly int SizeAsSourceOffset =
56                    ↳ Marshal.OffsetOf(typeof(Link), nameof(SizeAsSource)).ToInt32();
57                public static readonly int LeftAsTargetOffset =
58                    ↳ Marshal.OffsetOf(typeof(Link), nameof(LeftAsTarget)).ToInt32();
59                public static readonly int RightAsTargetOffset =
60                    ↳ Marshal.OffsetOf(typeof(Link),
61                        ↳ nameof(RightAsTarget)).ToInt32();
62                public static readonly int SizeAsTargetOffset =
63                    ↳ Marshal.OffsetOf(typeof(Link), nameof(SizeAsTarget)).ToInt32();
64
65                public TLink Source;
66                public TLink Target;
67            }
68        }
69    }

```

```

51     public TLink LeftAsSource;
52     public TLink RightAsSource;
53     public TLink SizeAsSource;
54     public TLink LeftAsTarget;
55     public TLink RightAsTarget;
56     public TLink SizeAsTarget;
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     public static TLink GetSource(IntPtr pointer) => (pointer +
60     ↪ SourceOffset).GetValue<TLink>();
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     public static TLink GetTarget(IntPtr pointer) => (pointer +
63     ↪ TargetOffset).GetValue<TLink>();
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     public static TLink GetLeftAsSource(IntPtr pointer) => (pointer +
66     ↪ LeftAsSourceOffset).GetValue<TLink>();
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     public static TLink GetRightAsSource(IntPtr pointer) => (pointer +
69     ↪ RightAsSourceOffset).GetValue<TLink>();
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     public static TLink GetSizeAsSource(IntPtr pointer) => (pointer +
72     ↪ SizeAsSourceOffset).GetValue<TLink>();
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public static TLink GetLeftAsTarget(IntPtr pointer) => (pointer +
75     ↪ LeftAsTargetOffset).GetValue<TLink>();
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     public static TLink GetRightAsTarget(IntPtr pointer) => (pointer +
78     ↪ RightAsTargetOffset).GetValue<TLink>();
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     public static TLink GetSizeAsTarget(IntPtr pointer) => (pointer +
81     ↪ SizeAsTargetOffset).GetValue<TLink>();
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     public static void SetSource(IntPtr pointer, TLink value) =>
85     ↪ (pointer + SourceOffset).SetValue(value);
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     public static void SetTarget(IntPtr pointer, TLink value) =>
88     ↪ (pointer + TargetOffset).SetValue(value);
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     public static void SetLeftAsSource(IntPtr pointer, TLink value) =>
91     ↪ (pointer + LeftAsSourceOffset).SetValue(value);
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public static void SetRightAsSource(IntPtr pointer, TLink value)
94     ↪ => (pointer + RightAsSourceOffset).SetValue(value);
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     public static void SetSizeAsSource(IntPtr pointer, TLink value) =>
97     ↪ (pointer + SizeAsSourceOffset).SetValue(value);
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     public static void SetLeftAsTarget(IntPtr pointer, TLink value) =>
100    ↪ (pointer + LeftAsTargetOffset).SetValue(value);
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     public static void SetRightAsTarget(IntPtr pointer, TLink value)
103     ↪ => (pointer + RightAsTargetOffset).SetValue(value);
104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
105     public static void SetSizeAsTarget(IntPtr pointer, TLink value) =>
106     ↪ (pointer + SizeAsTargetOffset).SetValue(value);
107
108     private struct LinksHeader
109     {
110         public static readonly int AllocatedLinksOffset =
111         ↪ Marshal.OffsetOf(typeof(LinksHeader),
112         ↪ nameof(AllocatedLinks)).ToInt32();
113     }

```

```

96     public static readonly int ReservedLinksOffset =
97     ↪ Marshal.OffsetOf(typeof(LinksHeader),
98     ↪ nameof(ReservedLinks)).ToInt32();
99     public static readonly int FreeLinksOffset =
100    ↪ Marshal.OffsetOf(typeof(LinksHeader),
101    ↪ nameof(FreeLinks)).ToInt32();
102     public static readonly int FirstFreeLinkOffset =
103    ↪ Marshal.OffsetOf(typeof(LinksHeader),
104    ↪ nameof(FirstFreeLink)).ToInt32();
105     public static readonly int FirstAsSourceOffset =
106    ↪ Marshal.OffsetOf(typeof(LinksHeader),
107    ↪ nameof(FirstAsSource)).ToInt32();
108     public static readonly int FirstAsTargetOffset =
109    ↪ Marshal.OffsetOf(typeof(LinksHeader),
110    ↪ nameof(FirstAsTarget)).ToInt32();
111     public static readonly int LastFreeLinkOffset =
112    ↪ Marshal.OffsetOf(typeof(LinksHeader),
113    ↪ nameof(LastFreeLink)).ToInt32();
114
115     public TLink AllocatedLinks;
116     public TLink ReservedLinks;
117     public TLink FreeLinks;
118     public TLink FirstFreeLink;
119     public TLink FirstAsSource;
120     public TLink FirstAsTarget;
121     public TLink LastFreeLink;
122     public TLink Reserved8;
123
124     [MethodImpl(MethodImplOptions.AggressiveInlining)]
125     public static TLink GetAllocatedLinks(IntPtr pointer) => (pointer
126     ↪ + AllocatedLinksOffset).GetValue<TLink>();
127     [MethodImpl(MethodImplOptions.AggressiveInlining)]
128     public static TLink GetReservedLinks(IntPtr pointer) => (pointer +
129     ↪ ReservedLinksOffset).GetValue<TLink>();
130     [MethodImpl(MethodImplOptions.AggressiveInlining)]
131     public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
132     ↪ FreeLinksOffset).GetValue<TLink>();
133     [MethodImpl(MethodImplOptions.AggressiveInlining)]
134     public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
135     ↪ FirstFreeLinkOffset).GetValue<TLink>();
136     [MethodImpl(MethodImplOptions.AggressiveInlining)]
137     public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
138     ↪ FirstAsSourceOffset).GetValue<TLink>();
139     [MethodImpl(MethodImplOptions.AggressiveInlining)]
140     public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
141     ↪ FirstAsTargetOffset).GetValue<TLink>();
142     [MethodImpl(MethodImplOptions.AggressiveInlining)]
143     public static TLink GetLastFreeLink(IntPtr pointer) => (pointer +
144     ↪ LastFreeLinkOffset).GetValue<TLink>();
145
146     [MethodImpl(MethodImplOptions.AggressiveInlining)]
147     public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) =>
148     ↪ pointer + FirstAsSourceOffset;
149     [MethodImpl(MethodImplOptions.AggressiveInlining)]
150     public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) =>
151     ↪ pointer + FirstAsTargetOffset;
152
153     [MethodImpl(MethodImplOptions.AggressiveInlining)]
154     public static void SetAllocatedLinks(IntPtr pointer, TLink value)
155     ↪ => (pointer + AllocatedLinksOffset).SetValue(value);
156     [MethodImpl(MethodImplOptions.AggressiveInlining)]
157     public static void SetReservedLinks(IntPtr pointer, TLink value)
158     ↪ => (pointer + ReservedLinksOffset).SetValue(value);
159     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

137     public static void SetFreeLinks(IntPtr pointer, TLink value) =>
138         ↳ (pointer + FreeLinksOffset).SetValue(value);
139     [MethodImpl(MethodImplOptions.AggressiveInlining)]
140     public static void SetFirstFreeLink(IntPtr pointer, TLink value)
141         ↳ => (pointer + FirstFreeLinkOffset).SetValue(value);
142     [MethodImpl(MethodImplOptions.AggressiveInlining)]
143     public static void SetFirstAsSource(IntPtr pointer, TLink value)
144         ↳ => (pointer + FirstAsSourceOffset).SetValue(value);
145     [MethodImpl(MethodImplOptions.AggressiveInlining)]
146     public static void SetFirstAsTarget(IntPtr pointer, TLink value)
147         ↳ => (pointer + FirstAsTargetOffset).SetValue(value);
148     [MethodImpl(MethodImplOptions.AggressiveInlining)]
149     public static void SetLastFreeLink(IntPtr pointer, TLink value) =>
150         ↳ (pointer + LastFreeLinkOffset).SetValue(value);
151 }
152
153 private readonly long _memoryReservationStep;
154
155 private readonly IResizableDirectMemory _memory;
156 private IntPtr _header;
157 private IntPtr _links;
158
159 private LinksTargetsTreeMethods _targetsTreeMethods;
160 private LinksSourcesTreeMethods _sourcesTreeMethods;
161
162 // TODO: Возможно чтобы гарантированно проверять на то, является ли
163 ↳ связь удалённой, нужно использовать не список а дерево, так как
164 ↳ так можно быстрее проверить на наличие связи внутри
165 private UnusedLinksListMethods _unusedLinksListMethods;
166
167 /// <summary>
168 /// Возвращает общее число связей находящихся в хранилище.
169 /// </summary>
170 private TLink Total =>
171     ↳ Subtract(LinksHeader.GetAllocatedLinks(_header),
172     ↳ LinksHeader.GetFreeLinks(_header));
173
174 public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
175
176 public ResizableDirectMemoryLinks(string address)
177     : this(address, DefaultLinksSizeStep)
178 {
179 }
180
181 /// <summary>
182 /// Создаёт экземпляр базы данных Links в файле по указанному адресу,
183 ↳ с указанным минимальным шагом расширения базы данных.
184 /// </summary>
185 /// <param name="address">Полный путь к файлу базы данных.</param>
186 /// <param name="memoryReservationStep">Минимальный шаг расширения
187 ↳ базы данных в байтах.</param>
188 public ResizableDirectMemoryLinks(string address, long
189     ↳ memoryReservationStep)
190     : this(new FileMappedResizableDirectMemory(address,
191     ↳ memoryReservationStep), memoryReservationStep)
192 {
193 }
194
195 public ResizableDirectMemoryLinks(IResizableDirectMemory memory)
196     : this(memory, DefaultLinksSizeStep)
197 {
198 }
199
200 public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
201     ↳ memoryReservationStep)
202 {
203 }

```

```

204 Constants = Default<LinksCombinedConstants<TLink, TLink,
205     ↳ int>>.Instance;
206 _memory = memory;
207 _memoryReservationStep = memoryReservationStep;
208 if (memory.ReservedCapacity < memoryReservationStep)
209 {
210     memory.ReservedCapacity = memoryReservationStep;
211 }
212 SetPointers(_memory);
213 // Гарантия корректности _memory.UsedCapacity относительно
214 ↳ _header->AllocatedLinks
215 _memory.UsedCapacity =
216     ↳ ((long)(Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)
217     ↳ * LinkSizeInBytes) + LinkHeaderSizeInBytes;
218 // Гарантия корректности _header->ReservedLinks относительно
219 ↳ _memory.ReservedCapacity
220 LinksHeader.SetReservedLinks(_header,
221     ↳ (Integer<TLink>)((_memory.ReservedCapacity -
222     ↳ LinkHeaderSizeInBytes) / LinkSizeInBytes));
223 }
224
225 [MethodImpl(MethodImplOptions.AggressiveInlining)]
226 public TLink Count(IList<TLink> restrictions)
227 {
228     // Если нет ограничений, тогда возвращаем общее число связей
229     ↳ находящихся в хранилище.
230     if (restrictions.Count == 0)
231     {
232         return Total;
233     }
234     if (restrictions.Count == 1)
235     {
236         var index = restrictions[Constants.IndexPart];
237         if (_equalityComparer.Equals(index, Constants.Any))
238         {
239             return Total;
240         }
241         return Exists(index) ? Integer<TLink>.One :
242             ↳ Integer<TLink>.Zero;
243     }
244     if (restrictions.Count == 2)
245     {
246         var index = restrictions[Constants.IndexPart];
247         var value = restrictions[1];
248         if (_equalityComparer.Equals(index, Constants.Any))
249         {
250             if (_equalityComparer.Equals(value, Constants.Any))
251             {
252                 return Total; // Any - как отсутствие ограничения
253             }
254             return Add(_sourcesTreeMethods.CalculateReferences(value),
255                 ↳ _targetsTreeMethods.CalculateReferences(value));
256         }
257         else
258         {
259             if (!Exists(index))
260             {
261                 return Integer<TLink>.Zero;
262             }
263             if (_equalityComparer.Equals(value, Constants.Any))
264             {
265                 return Integer<TLink>.One;
266             }
267             var storedLinkValue = GetLinkUnsafe(index);

```

```

243         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value)
244             ||
245             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
246         {
247             return Integer<TLink>.One;
248         }
249         return Integer<TLink>.Zero;
250     }
251     if (restrictions.Count == 3)
252     {
253         var index = restrictions[Constants.IndexPart];
254         var source = restrictions[Constants.SourcePart];
255         var target = restrictions[Constants.TargetPart];
256
257         if (_equalityComparer.Equals(index, Constants.Any))
258         {
259             if (_equalityComparer.Equals(source, Constants.Any) &&
260                 _equalityComparer.Equals(target, Constants.Any))
261             {
262                 return Total;
263             }
264             else if (_equalityComparer.Equals(source, Constants.Any))
265             {
266                 return _targetsTreeMethods.CalculateReferences(target);
267             }
268             else if (_equalityComparer.Equals(target, Constants.Any))
269             {
270                 return _sourcesTreeMethods.CalculateReferences(source);
271             }
272             else //if(source != Any && target != Any)
273             {
274                 // Эквивалент Exists(source, target) => Count(Any,
275                 //     source, target) > 0
276                 var link = _sourcesTreeMethods.Search(source, target);
277                 return _equalityComparer.Equals(link, Constants.Null)
278                     ? Integer<TLink>.Zero : Integer<TLink>.One;
279             }
280         }
281     }
282     else
283     {
284         if (!Exists(index))
285         {
286             return Integer<TLink>.Zero;
287         }
288         if (_equalityComparer.Equals(source, Constants.Any) &&
289             _equalityComparer.Equals(target, Constants.Any))
290         {
291             return Integer<TLink>.One;
292         }
293         var storedLinkValue = GetLinkUnsafe(index);
294         if (!_equalityComparer.Equals(source, Constants.Any) &&
295             !_equalityComparer.Equals(target, Constants.Any))
296         {
297             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source)
298                 &&
299                 _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
300             {
301                 return Integer<TLink>.One;
302             }
303         }
304     }
305 }

```

```

295     }
296     return Integer<TLink>.Zero;
297 }
298 var value = default(TLink);
299 if (_equalityComparer.Equals(source, Constants.Any))
300 {
301     value = target;
302 }
303 if (_equalityComparer.Equals(target, Constants.Any))
304 {
305     value = source;
306 }
307 if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value)
308     ||
309     _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
310 {
311     return Integer<TLink>.One;
312 }
313 return Integer<TLink>.Zero;
314 }
315 }
316 throw new NotSupportedException("Другие размеры и способы
317     ограничений не поддерживаются.");
318 }
319
320 [MethodImpl(MethodImplOptions.AggressiveInlining)]
321 public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink>
322     restrictions)
323 {
324     if (restrictions.Count == 0)
325     {
326         for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
327             (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <=
328             0; link = Increment(link))
329         {
330             if (Exists(link) &&
331                 _equalityComparer.Equals(handler(GetLinkStruct(link)),
332                     Constants.Break))
333             {
334                 return Constants.Break;
335             }
336         }
337         return Constants.Continue;
338     }
339     if (restrictions.Count == 1)
340     {
341         var index = restrictions[Constants.IndexPart];
342         if (_equalityComparer.Equals(index, Constants.Any))
343         {
344             return Each(handler, ArrayPool<TLink>.Empty);
345         }
346         if (!Exists(index))
347         {
348             return Constants.Continue;
349         }
350         return handler(GetLinkStruct(index));
351     }
352     if (restrictions.Count == 2)
353     {
354         var index = restrictions[Constants.IndexPart];
355         var value = restrictions[1];
356     }
357 }

```



```

350 if (_equalityComparer.Equals(index, Constants.Any))
351 {
352     if (_equalityComparer.Equals(value, Constants.Any))
353     {
354         return Each(handler, ArrayPool<TLink>.Empty);
355     }
356     if (_equalityComparer.Equals(Each(handler, new[] { index,
357 ↪ value, Constants.Any })), Constants.Break))
358     {
359         return Constants.Break;
360     }
361     return Each(handler, new[] { index, Constants.Any, value
362 ↪ });
363 }
364 else
365 {
366     if (!Exists(index))
367     {
368         return Constants.Continue;
369     }
370     if (_equalityComparer.Equals(value, Constants.Any))
371     {
372         return handler(GetLinkStruct(index));
373     }
374     var storedLinkValue = GetLinkUnsafe(index);
375     if (_equalityComparer.Equals(Link.GetSource(storedLinkValu
376 ↪ e), value)
377     {
378         if (_equalityComparer.Equals(Link.GetTarget(storedLinkValu
379 ↪ e),
380 ↪ value))
381     {
382         return handler(GetLinkStruct(index));
383     }
384     return Constants.Continue;
385 }
386 }
387 if (restrictions.Count == 3)
388 {
389     var index = restrictions[Constants.IndexPart];
390     var source = restrictions[Constants.SourcePart];
391     var target = restrictions[Constants.TargetPart];
392     if (_equalityComparer.Equals(index, Constants.Any))
393     {
394         if (_equalityComparer.Equals(source, Constants.Any) &&
395 ↪ _equalityComparer.Equals(target, Constants.Any))
396     {
397         return Each(handler, ArrayPool<TLink>.Empty);
398     }
399     else if (_equalityComparer.Equals(source, Constants.Any))
400     {
401         return _targetsTreeMethods.EachReference(target,
402 ↪ handler);
403     }
404     else if (_equalityComparer.Equals(target, Constants.Any))
405     {
406         return _sourcesTreeMethods.EachReference(source,
407 ↪ handler);
408     }
409     else //if(source != Any && target != Any)
410     {
411         var link = _sourcesTreeMethods.Search(source, target);
412         return _equalityComparer.Equals(link, Constants.Null)
413 ↪ ? Constants.Continue :
414 ↪ handler(GetLinkStruct(link));
415     }
416 }

```

```

404 }
405 }
406 else
407 {
408     if (!Exists(index))
409     {
410         return Constants.Continue;
411     }
412     if (_equalityComparer.Equals(source, Constants.Any) &&
413 ↪ _equalityComparer.Equals(target, Constants.Any))
414     {
415         return handler(GetLinkStruct(index));
416     }
417     var storedLinkValue = GetLinkUnsafe(index);
418     if (_equalityComparer.Equals(source, Constants.Any) &&
419 ↪ !_equalityComparer.Equals(target, Constants.Any))
420     {
421         if (_equalityComparer.Equals(Link.GetSource(storedLink
422 ↪ Value), source)
423 ↪ &&
424 ↪ _equalityComparer.Equals(Link.GetTarget(storedLink
425 ↪ Value),
426 ↪ target))
427     {
428         return handler(GetLinkStruct(index));
429     }
430     return Constants.Continue;
431 }
432 var value = default(TLink);
433 if (_equalityComparer.Equals(source, Constants.Any))
434 {
435     value = target;
436 }
437 if (_equalityComparer.Equals(target, Constants.Any))
438 {
439     value = source;
440 }
441 if (_equalityComparer.Equals(Link.GetSource(storedLinkValu
442 ↪ e), value)
443 ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValu
444 ↪ e),
445 ↪ value))
446     {
447         return handler(GetLinkStruct(index));
448     }
449     return Constants.Continue;
450 }
451 }
452 throw new NotSupportedException("Другие размеры и способы
453 ↪ ограничений не поддерживаются.");
454 }
455
456 /// <remarks>
457 /// TODO: Возможно можно перемещать значения, если указан индекс, но
458 ↪ значение существует в другом месте (но не в менеджере памяти, а в
459 ↪ логике Links)
460 /// </remarks>
461 [MethodImpl(MethodImplOptions.AggressiveInlining)]
462 public TLink Update(IList<TLink> values)
463 {
464     var linkIndex = values[Constants.IndexPart];
465     var link = GetLinkUnsafe(linkIndex);

```

```

454 // Будет корректно работать только в том случае, если пространство
455 ↪ выделенной связи предварительно заполнено нулями
456 if (!_equalityComparer.Equals(Link.GetSource(link),
457 ↪ Constants.Null))
458 {
459     _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer ↪
460     ↪ (_header),
461     ↪ linkIndex);
462 }
463 Link.SetSource(link, values[Constants.SourcePart]);
464 Link.SetTarget(link, values[Constants.TargetPart]);
465 if (!_equalityComparer.Equals(Link.GetSource(link),
466 ↪ Constants.Null))
467 {
468     _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer ↪
469     ↪ (_header),
470     ↪ linkIndex);
471 }
472 }
473 return linkIndex;
474 }
475
476 [MethodImpl(MethodImplOptions.AggressiveInlining)]
477 public Link<TLink> GetLinkStruct(TLink linkIndex)
478 {
479     var link = GetLinkUnsafe(linkIndex);
480     return new Link<TLink>(linkIndex, Link.GetSource(link),
481     ↪ Link.GetTarget(link));
482 }
483
484 [MethodImpl(MethodImplOptions.AggressiveInlining)]
485 private IntPtr GetLinkUnsafe(TLink linkIndex) =>
486 ↪ _links.GetElement(LinkSizeInBytes, linkIndex);
487
488 /// <remarks>
489 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими
490 ↪ не заполняет пространство
491 /// </remarks>
492 public TLink Create()
493 {
494     var freeLink = LinksHeader.GetFirstFreeLink(_header);
495     if (!_equalityComparer.Equals(freeLink, Constants.Null))
496     {
497         _unusedLinksListMethods.Detach(freeLink);
498     }
499     else
500     {
501         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
502     ↪ Constants.MaxPossibleIndex) > 0)
503         {

```

```

504     throw new LinksLimitReachedException((Integer<TLink>)Const ↪
505     ↪ ants.MaxPossibleIndex);
506 }
507 }
508 if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
509 ↪ Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
510 {
511     _memory.ReservedCapacity += _memoryReservationStep;
512     SetPointers(_memory);
513     LinksHeader.SetReservedLinks(_header,
514     ↪ (Integer<TLink>)(_memory.ReservedCapacity /
515     ↪ LinkSizeInBytes));
516 }
517 LinksHeader.SetAllocatedLinks(_header,
518     ↪ Increment(LinksHeader.GetAllocatedLinks(_header)));
519 _memory.UsedCapacity += LinkSizeInBytes;
520 freeLink = LinksHeader.GetAllocatedLinks(_header);
521 }
522 return freeLink;
523 }
524
525 public void Delete(TLink link)
526 {
527     if (_comparer.Compare(link,
528     ↪ LinksHeader.GetAllocatedLinks(_header)) < 0)
529     {
530         _unusedLinksListMethods.AttachAsFirst(link);
531     }
532     else if (_equalityComparer.Equals(link,
533     ↪ LinksHeader.GetAllocatedLinks(_header)))
534     {
535         LinksHeader.SetAllocatedLinks(_header,
536     ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
537         _memory.UsedCapacity -= LinkSizeInBytes;
538         // Убираем все связи, находящиеся в списке свободных в конце
539         ↪ файла, до тех пор, пока не дойдём до первой существующей
540         ↪ связи
541         // Позволяет оптимизировать количество выделенных связей
542         ↪ (AllocatedLinks)
543         while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_heade ↪
544         ↪ r), Integer<TLink>.Zero) > 0) &&
545         ↪ IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
546         {
547             _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLin ↪
548             ↪ ks(_header));
549             LinksHeader.SetAllocatedLinks(_header,
550             ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
551             _memory.UsedCapacity -= LinkSizeInBytes;
552         }
553     }
554 }
555
556 /// <remarks>
557 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в
558     ↪ том случае, если адрес реально поменялся
559 ///
560 /// Указатель this.links может быть в том же месте,
561     ↪ так как 0-я связь не используется и имеет такой же размер как
562     ↪ Header,
563     ↪ поэтому header размещается в том же месте, что и 0-я связь
564     ↪ </remarks>
565 private void SetPointers(IDirectMemory memory)
566 {
567     if (memory == null)
568     {

```

```

547     _links = IntPtr.Zero;
548     _header = _links;
549     _unusedLinksListMethods = null;
550     _targetsTreeMethods = null;
551     _unusedLinksListMethods = null;
552 }
553 else
554 {
555     _links = memory.Pointer;
556     _header = _links;
557     _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
558     _targetsTreeMethods = new LinksTargetsTreeMethods(this);
559     _unusedLinksListMethods = new UnusedLinksListMethods(_links,
560         ↪ _header);
561 }
562
563 [MethodImpl(MethodImplOptions.AggressiveInlining)]
564 private bool Exists(TLink link)
565 => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
566 && (_comparer.Compare(link,
567     ↪ LinksHeader.GetAllocatedLinks(_header)) <= 0)
568 && !IsUnusedLink(link);
569
570 [MethodImpl(MethodImplOptions.AggressiveInlining)]
571 private bool IsUnusedLink(TLink link)
572 => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header),
573     ↪ link)
574 || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)),
575     ↪ Constants.Null)
576 && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)),
577     ↪ Constants.Null));
578
579 #region DisposableBase
580
581 protected override bool AllowMultipleDisposeCalls => true;
582
583 protected override void DisposeCore(bool manual, bool wasDisposed)
584 {
585     if (!wasDisposed)
586     {
587         SetPointers(null);
588     }
589     Disposable.TryDispose(_memory);
590 }
591
592 #endregion
593 }
594 }

```

./ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs

```

1  using System;
2  using Platform.Unsafe;
3  using Platform.Collections.Methods.Lists;
4
5  namespace Platform.Data.Doublets.ResizableDirectMemory
6  {
7      partial class ResizableDirectMemoryLinks<TLink>
8      {
9          private class UnusedLinksListMethods :
10             ↪ CircularDoublyLinkedListMethods<TLink>
11          {
12              private readonly IntPtr _links;
13              private readonly IntPtr _header;
14
15              public UnusedLinksListMethods(IntPtr links, IntPtr header)

```

```

15  {
16      _links = links;
17      _header = header;
18  }
19
20  protected override TLink GetFirst() => (_header +
21     ↪ LinksHeader.FirstFreeLinkOffset).GetValue<TLink>();
22
23  protected override TLink GetLast() => (_header +
24     ↪ LinksHeader.LastFreeLinkOffset).GetValue<TLink>();
25
26  protected override TLink GetPrevious(TLink element) =>
27     ↪ (_links.GetElement(LinkSizeInBytes, element) +
28     ↪ Link.SourceOffset).GetValue<TLink>();
29
30  protected override TLink GetNext(TLink element) =>
31     ↪ (_links.GetElement(LinkSizeInBytes, element) +
32     ↪ Link.TargetOffset).GetValue<TLink>();
33
34  protected override TLink GetSize() => (_header +
35     ↪ LinksHeader.FreeLinksOffset).GetValue<TLink>();
36
37  protected override void SetFirst(TLink element) => (_header +
38     ↪ LinksHeader.FirstFreeLinkOffset).SetValue(element);
39
40  protected override void SetLast(TLink element) => (_header +
41     ↪ LinksHeader.LastFreeLinkOffset).SetValue(element);
42
43  protected override void SetPrevious(TLink element, TLink previous)
44     ↪ => (_links.GetElement(LinkSizeInBytes, element) +
45     ↪ Link.SourceOffset).SetValue(previous);
46
47  protected override void SetNext(TLink element, TLink next) =>
48     ↪ (_links.GetElement(LinkSizeInBytes, element) +
49     ↪ Link.TargetOffset).SetValue(next);
50
51  protected override void SetSize(TLink size) => (_header +
52     ↪ LinksHeader.FreeLinksOffset).SetValue(size);
53
54  }
55  }

```

./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Numbers;
6  using Platform.Unsafe;
7  using Platform.Collections.Methods.Trees;
8  using Platform.Data.Constants;
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory
11 {
12     partial class ResizableDirectMemoryLinks<TLink>
13     {
14         private abstract class LinksTreeMethodsBase :
15             ↪ SizedAndThreadedAVLBalancedTreeMethods<TLink>
16         {
17             private readonly ResizableDirectMemoryLinks<TLink> _memory;
18             private readonly LinksCombinedConstants<TLink, TLink, int>
19                 ↪ _constants;
20             protected readonly IntPtr Links;
21             protected readonly IntPtr Header;

```

```

21     protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink>
22         ↪ memory)
23     {
24         Links = memory._links;
25         Header = memory._header;
26         _memory = memory;
27         _constants = memory.Constants;
28     }
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected abstract TLink GetTreeRoot();
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected abstract TLink GetBasePartValue(TLink link);
35
36     public TLink this[TLink index]
37     {
38         get
39         {
40             var root = GetTreeRoot();
41             if (GreaterOrEqualThan(index, GetSize(root)))
42             {
43                 return GetZero();
44             }
45             while (!EqualToZero(root))
46             {
47                 var left = GetLeftOrDefault(root);
48                 var leftSize = GetSizeOrZero(left);
49                 if (LessThan(index, leftSize))
50                 {
51                     root = left;
52                     continue;
53                 }
54                 if (IsEquals(index, leftSize))
55                 {
56                     return root;
57                 }
58                 root = GetRightOrDefault(root);
59                 index = Subtract(index, Increment(leftSize));
60             }
61             return GetZero(); // TODO: Impossible situation exception
62             ↪ (only if tree structure broken)
63         }
64     }
65
66     // TODO: Return indices range instead of references count
67     public TLink CalculateReferences(TLink link)
68     {
69         var root = GetTreeRoot();
70         var total = GetSize(root);
71         var totalRightIgnore = GetZero();
72         while (!EqualToZero(root))
73         {
74             var @base = GetBasePartValue(root);
75             if (LessOrEqualThan(@base, link))
76             {
77                 root = GetRightOrDefault(root);
78             }
79             else
80             {
81                 totalRightIgnore = Add(totalRightIgnore,
82                     ↪ Increment(GetRightSize(root)));
83                 root = GetLeftOrDefault(root);
84             }
85         }
86         root = GetTreeRoot();

```

```

84     var totalLeftIgnore = GetZero();
85     while (!EqualToZero(root))
86     {
87         var @base = GetBasePartValue(root);
88         if (GreaterOrEqualThan(@base, link))
89         {
90             root = GetLeftOrDefault(root);
91         }
92         else
93         {
94             totalLeftIgnore = Add(totalLeftIgnore,
95                 ↪ Increment(GetLeftSize(root)));
96             root = GetRightOrDefault(root);
97         }
98     }
99     return Subtract(Subtract(total, totalRightIgnore),
100         ↪ totalLeftIgnore);
101 }
102
103 public TLink EachReference(TLink link, Func<IList<TLink>, TLink>
104     ↪ handler)
105 {
106     var root = GetTreeRoot();
107     if (EqualToZero(root))
108     {
109         return _constants.Continue;
110     }
111     TLink first = GetZero(), current = root;
112     while (!EqualToZero(current))
113     {
114         var @base = GetBasePartValue(current);
115         if (GreaterOrEqualThan(@base, link))
116         {
117             if (IsEquals(@base, link))
118             {
119                 first = current;
120             }
121             current = GetLeftOrDefault(current);
122         }
123         else
124         {
125             current = GetRightOrDefault(current);
126         }
127     }
128     if (!EqualToZero(first))
129     {
130         current = first;
131         while (true)
132         {
133             if (IsEquals(handler(_memory.GetLinkStruct(current)),
134                 ↪ _constants.Break))
135             {
136                 return _constants.Break;
137             }
138             current = GetNext(current);
139             if (EqualToZero(current) ||
140                 ↪ !IsEquals(GetBasePartValue(current), link))
141             {
142                 break;
143             }
144         }
145     }
146     return _constants.Continue;
147 }

```

```

144     protected override void PrintNodeValue(TLink node, StringBuilder
145     ↪ sb)
146     {
147         sb.Append(' ');
148         sb.Append((Links.GetElement(LinkSizeInBytes, node) +
149         ↪ Link.SourceOffset).GetValue<TLink>());
150         sb.Append('-');
151         sb.Append('>');
152         sb.Append((Links.GetElement(LinkSizeInBytes, node) +
153         ↪ Link.TargetOffset).GetValue<TLink>());
154     }
155 }
156 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
157 {
158     public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink>
159     ↪ memory)
160     : base(memory)
161     {
162     }
163
164     protected override IntPtr GetLeftPointer(TLink node) =>
165     ↪ Links.GetElement(LinkSizeInBytes, node) +
166     ↪ Link.LeftAsSourceOffset;
167
168     protected override IntPtr GetRightPointer(TLink node) =>
169     ↪ Links.GetElement(LinkSizeInBytes, node) +
170     ↪ Link.RightAsSourceOffset;
171
172     protected override TLink GetLeftValue(TLink node) =>
173     ↪ (Links.GetElement(LinkSizeInBytes, node) +
174     ↪ Link.LeftAsSourceOffset).GetValue<TLink>();
175
176     protected override TLink GetRightValue(TLink node) =>
177     ↪ (Links.GetElement(LinkSizeInBytes, node) +
178     ↪ Link.RightAsSourceOffset).GetValue<TLink>();
179
180     protected override TLink GetSize(TLink node)
181     {
182         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
183         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
184         return BitwiseHelpers.PartialRead(previousValue, 5, -5);
185     }
186
187     protected override void SetLeft(TLink node, TLink left) =>
188     ↪ (Links.GetElement(LinkSizeInBytes, node) +
189     ↪ Link.LeftAsSourceOffset).SetValue(left);
190
191     protected override void SetRight(TLink node, TLink right) =>
192     ↪ (Links.GetElement(LinkSizeInBytes, node) +
193     ↪ Link.RightAsSourceOffset).SetValue(right);
194
195     protected override void SetSize(TLink node, TLink size)
196     {
197         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
198         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
199         (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsSourceOf
200         ↪ fset).SetValue(BitwiseHelpers.PartialWrite(previousValue,
201         ↪ size, 5, -5));
202     }
203
204     protected override bool GetLeftIsChild(TLink node)
205     {
206     }
207 }

```

```

188     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
189     ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
190     return
191     ↪ (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue,
192     ↪ 4, 1);
193 }
194
195     protected override void SetLeftIsChild(TLink node, bool value)
196     {
197         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
198         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
199         var modified = BitwiseHelpers.PartialWrite(previousValue,
200         ↪ (TLink)(Integer<TLink>)value, 4, 1);
201         (Links.GetElement(LinkSizeInBytes, node) +
202         ↪ Link.SizeAsSourceOffset).SetValue(modified);
203     }
204
205     protected override bool GetRightIsChild(TLink node)
206     {
207         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
208         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
209         return
210         ↪ (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue,
211         ↪ 3, 1);
212     }
213
214     protected override void SetRightIsChild(TLink node, bool value)
215     {
216         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
217         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
218         var modified = BitwiseHelpers.PartialWrite(previousValue,
219         ↪ (TLink)(Integer<TLink>)value, 3, 1);
220         (Links.GetElement(LinkSizeInBytes, node) +
221         ↪ Link.SizeAsSourceOffset).SetValue(modified);
222     }
223
224     protected override sbyte GetBalance(TLink node)
225     {
226         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
227         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
228         var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(
229         ↪ previousValue, 0,
230         ↪ 3);
231         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) <<
232         ↪ 5) | value & 3 | 124 : value & 3);
233         return unpackedValue;
234     }
235
236     protected override void SetBalance(TLink node, sbyte value)
237     {
238         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
239         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
240         var packagedValue = (TLink)(Integer<TLink>)(((byte)value >>
241         ↪ 5) & 4) | value & 3);
242         var modified = BitwiseHelpers.PartialWrite(previousValue,
243         ↪ packagedValue, 0, 3);
244         (Links.GetElement(LinkSizeInBytes, node) +
245         ↪ Link.SizeAsSourceOffset).SetValue(modified);
246     }
247
248     protected override bool FirstIsToTheLeftOfSecond(TLink first,
249     ↪ TLink second)
250     {
251     }

```

```

230     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
231         ↪ Link.SourceOffset).GetValue<TLink>();
232     var secondSource = (Links.GetElement(LinkSizeInBytes, second)
233         ↪ + Link.SourceOffset).GetValue<TLink>();
234     return LessThan(firstSource, secondSource) ||
235         (IsEquals(firstSource, secondSource) &&
236             ↪ LessThan((Links.GetElement(LinkSizeInBytes, first)
237                 ↪ + Link.TargetOffset).GetValue<TLink>(),
238                 ↪ (Links.GetElement(LinkSizeInBytes, second) +
239                 ↪ Link.TargetOffset).GetValue<TLink>()));
240 }
241
242 protected override bool FirstIsToTheRightOfSecond(TLink first,
243     ↪ TLink second)
244 {
245     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
246         ↪ Link.SourceOffset).GetValue<TLink>();
247     var secondSource = (Links.GetElement(LinkSizeInBytes, second)
248         ↪ + Link.SourceOffset).GetValue<TLink>();
249     return GreaterThan(firstSource, secondSource) ||
250         (IsEquals(firstSource, secondSource) &&
251             ↪ GreaterThan((Links.GetElement(LinkSizeInBytes,
252                 ↪ first) + Link.TargetOffset).GetValue<TLink>(),
253                 ↪ (Links.GetElement(LinkSizeInBytes, second) +
254                 ↪ Link.TargetOffset).GetValue<TLink>()));
255 }
256
257 protected override TLink GetTreeRoot() => (Header +
258     ↪ LinksHeader.FirstAsSourceOffset).GetValue<TLink>();
259
260 protected override TLink GetBasePartValue(TLink link) =>
261     (Links.GetElement(LinkSizeInBytes, link) +
262     ↪ Link.SourceOffset).GetValue<TLink>();
263
264 /// <summary>
265 /// Выполняет поиск и возвращает индекс связи с указанными Source
266 ↪ (началом) и Target (концом)
267 /// по дереву (индексу) связей, отсортированному по Source, а
268 ↪ затем по Target.
269 /// </summary>
270 /// <param name="source">Индекс связи, которая является началом на
271 ↪ искомой связи.</param>
272 /// <param name="target">Индекс связи, которая является концом на
273 ↪ искомой связи.</param>
274 /// <returns>Индекс искомой связи.</returns>
275 public TLink Search(TLink source, TLink target)
276 {
277     var root = GetTreeRoot();
278     while (!EqualToZero(root))
279     {
280         var rootSource = (Links.GetElement(LinkSizeInBytes, root)
281             ↪ + Link.SourceOffset).GetValue<TLink>();
282         var rootTarget = (Links.GetElement(LinkSizeInBytes, root)
283             ↪ + Link.TargetOffset).GetValue<TLink>();
284         if (FirstIsToTheLeftOfSecond(source, target, rootSource,
285             ↪ rootTarget)) // node.Key < root.Key
286         {
287             root = GetLeftOrDefault(root);
288         }
289         else if (FirstIsToTheRightOfSecond(source, target,
290             ↪ rootSource, rootTarget)) // node.Key > root.Key
291         {
292             root = GetRightOrDefault(root);
293         }
294     }
295 }

```

```

269     }
270     else // node.Key == root.Key
271     {
272         return root;
273     }
274 }
275 return GetZero();
276 }
277
278 [MethodImpl(MethodImplOptions.AggressiveInlining)]
279 private bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink
280     ↪ firstTarget, TLink secondSource, TLink secondTarget) =>
281     LessThan(firstSource, secondSource) || (IsEquals(firstSource,
282     ↪ secondSource) && LessThan(firstTarget, secondTarget));
283
284 [MethodImpl(MethodImplOptions.AggressiveInlining)]
285 private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink
286     ↪ firstTarget, TLink secondSource, TLink secondTarget) =>
287     GreaterThan(firstSource, secondSource) ||
288     (IsEquals(firstSource, secondSource) &&
289     ↪ GreaterThan(firstTarget, secondTarget));
290 }
291
292 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
293 {
294     public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink>
295     ↪ memory)
296     : base(memory)
297     {
298     }
299
300     protected override IntPtr GetLeftPointer(TLink node) =>
301     ↪ Links.GetElement(LinkSizeInBytes, node) +
302     ↪ Link.LeftAsTargetOffset;
303
304     protected override IntPtr GetRightPointer(TLink node) =>
305     ↪ Links.GetElement(LinkSizeInBytes, node) +
306     ↪ Link.RightAsTargetOffset;
307
308     protected override TLink GetLeftValue(TLink node) =>
309     ↪ (Links.GetElement(LinkSizeInBytes, node) +
310     ↪ Link.LeftAsTargetOffset).GetValue<TLink>();
311
312     protected override TLink GetRightValue(TLink node) =>
313     ↪ (Links.GetElement(LinkSizeInBytes, node) +
314     ↪ Link.RightAsTargetOffset).GetValue<TLink>();
315
316     protected override TLink GetSize(TLink node)
317     {
318         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
319             ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
320         return BitwiseHelpers.PartialRead(previousValue, 5, -5);
321     }
322
323     protected override void SetLeft(TLink node, TLink left) =>
324     ↪ (Links.GetElement(LinkSizeInBytes, node) +
325     ↪ Link.LeftAsTargetOffset).SetValue(left);
326
327     protected override void SetRight(TLink node, TLink right) =>
328     ↪ (Links.GetElement(LinkSizeInBytes, node) +
329     ↪ Link.RightAsTargetOffset).SetValue(right);
330
331     protected override void SetSize(TLink node, TLink size)
332     {

```

```

312     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
313         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
314     (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsTargetOf
315         ↪ fset).SetValue(BitwiseHelpers.PartialWrite(previousValue,
316         ↪ size, 5, -5));
317 }
318
319 protected override bool GetLeftIsChild(TLink node)
320 {
321     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
322         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
323     return
324         ↪ (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue,
325         ↪ 4, 1);
326 }
327
328 protected override void SetLeftIsChild(TLink node, bool value)
329 {
330     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
331         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
332     var modified = BitwiseHelpers.PartialWrite(previousValue,
333         ↪ (TLink)(Integer<TLink>)value, 4, 1);
334     (Links.GetElement(LinkSizeInBytes, node) +
335         ↪ Link.SizeAsTargetOffset).SetValue(modified);
336 }
337
338 protected override bool GetRightIsChild(TLink node)
339 {
340     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
341         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
342     return
343         ↪ (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue,
344         ↪ 3, 1);
345 }
346
347 protected override void SetRightIsChild(TLink node, bool value)
348 {
349     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
350         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
351     var modified = BitwiseHelpers.PartialWrite(previousValue,
352         ↪ (TLink)(Integer<TLink>)value, 3, 1);
353     (Links.GetElement(LinkSizeInBytes, node) +
354         ↪ Link.SizeAsTargetOffset).SetValue(modified);
355 }
356
357 protected override sbyte GetBalance(TLink node)
358 {
359     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
360         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
361     var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(
362         ↪ previousValue, 0,
363         ↪ 3);
364     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) <<
365         ↪ 5) | value & 3 | 124 : value & 3);
366     return unpackedValue;
367 }
368
369 protected override void SetBalance(TLink node, sbyte value)
370 {
371     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
372         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
373     var packagedValue = (TLink)(Integer<TLink>)(((byte)value >>
374         ↪ 5) & 4) | value & 3);
375 }

```

```

354     var modified = BitwiseHelpers.PartialWrite(previousValue,
355         ↪ packagedValue, 0, 3);
356     (Links.GetElement(LinkSizeInBytes, node) +
357         ↪ Link.SizeAsTargetOffset).SetValue(modified);
358 }
359
360 protected override bool FirstIsToTheLeftOfSecond(TLink first,
361     ↪ TLink second)
362 {
363     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
364         ↪ Link.TargetOffset).GetValue<TLink>();
365     var secondTarget = (Links.GetElement(LinkSizeInBytes, second)
366         ↪ + Link.TargetOffset).GetValue<TLink>();
367     return LessThan(firstTarget, secondTarget) ||
368         (IsEquals(firstTarget, secondTarget) &&
369         ↪ LessThan((Links.GetElement(LinkSizeInBytes, first)
370         ↪ + Link.SourceOffset).GetValue<TLink>(),
371         ↪ (Links.GetElement(LinkSizeInBytes, second) +
372         ↪ Link.SourceOffset).GetValue<TLink>()));
373 }
374
375 protected override bool FirstIsToTheRightOfSecond(TLink first,
376     ↪ TLink second)
377 {
378     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
379         ↪ Link.TargetOffset).GetValue<TLink>();
380     var secondTarget = (Links.GetElement(LinkSizeInBytes, second)
381         ↪ + Link.TargetOffset).GetValue<TLink>();
382     return GreaterThan(firstTarget, secondTarget) ||
383         (IsEquals(firstTarget, secondTarget) &&
384         ↪ GreaterThan((Links.GetElement(LinkSizeInBytes,
385         ↪ first) + Link.SourceOffset).GetValue<TLink>(),
386         ↪ (Links.GetElement(LinkSizeInBytes, second) +
387         ↪ Link.SourceOffset).GetValue<TLink>()));
388 }
389
390 protected override TLink GetTreeRoot() => (Header +
391     ↪ LinksHeader.FirstAsTargetOffset).GetValue<TLink>();
392
393 protected override TLink GetBasePartValue(TLink link) =>
394     ↪ (Links.GetElement(LinkSizeInBytes, link) +
395     ↪ Link.TargetOffset).GetValue<TLink>();
396 }
397 }
398 }
399 }

```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Collections.Arrays;
6  using Platform.Helpers.Singletons;
7  using Platform.Memory;
8  using Platform.Data.Exceptions;
9  using Platform.Data.Constants;
10
11  // #define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
12
13  #pragma warning disable 0649
14  #pragma warning disable 169
15
16  // ReSharper disable BuiltInTypeReferenceStyle
17
18  namespace Platform.Data.Doublets.ResizableDirectMemory

```

```

19 {
20     using id = UInt64;
21
22     public unsafe partial class UInt64ResizableDirectMemoryLinks :
23     ↪ DisposableBase, ILinks<id>
24     {
25         /// <summary>Возвращает размер одной связи в байтах.</summary>
26         /// <remarks>
27         /// Используется только во вне класса, не рекомендуется использовать
28         ↪ внутри.
29         /// Так как во вне не обязательно будет доступен unsafe C#.
30         /// </remarks>
31         public static readonly int LinkSizeInBytes = sizeof(Link);
32
33         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes *
34         ↪ 1024 * 1024;
35
36         private struct Link
37         {
38             public id Source;
39             public id Target;
40             public id LeftAsSource;
41             public id RightAsSource;
42             public id SizeAsSource;
43             public id LeftAsTarget;
44             public id RightAsTarget;
45             public id SizeAsTarget;
46         }
47
48         private struct LinksHeader
49         {
50             public id AllocatedLinks;
51             public id ReservedLinks;
52             public id FreeLinks;
53             public id FirstFreeLink;
54             public id FirstAsSource;
55             public id FirstAsTarget;
56             public id LastFreeLink;
57             public id Reserved8;
58         }
59
60         private readonly long _memoryReservationStep;
61
62         private readonly IResizableDirectMemory _memory;
63         private LinksHeader* _header;
64         private Link* _links;
65
66         private LinksTargetsTreeMethods _targetsTreeMethods;
67         private LinksSourcesTreeMethods _sourcesTreeMethods;
68
69         // TODO: Возможно чтобы гарантированно проверять на то, является ли
70         ↪ связь удалённой, нужно использовать не список а дерево, так как
71         ↪ так можно быстрее проверить на наличие связи внутри
72         private UnusedLinksListMethods _unusedLinksListMethods;
73
74         /// <summary>
75         /// Возвращает общее число связей находящихся в хранилище.
76         /// </summary>
77         private id Total => _header->AllocatedLinks - _header->FreeLinks;
78
79         // TODO: Дать возможность переопределять в конструкторе
80         public LinksCombinedConstants<id, id, int> Constants { get; }
81
82         public UInt64ResizableDirectMemoryLinks(string address) :
83         ↪ this(address, DefaultLinksSizeStep) { }
84
85         /// <summary>
86         /// Создаёт экземпляр базы данных Links в файле по указанному адресу,
87         ↪ с указанным минимальным шагом расширения базы данных.

```

```

81     /// </summary>
82     /// <param name="address">Полный путь к файлу базы данных.</param>
83     /// <param name="memoryReservationStep">Минимальный шаг расширения
84     ↪ базы данных в байтах.</param>
85     public UInt64ResizableDirectMemoryLinks(string address, long
86     ↪ memoryReservationStep) : this(new
87     ↪ FileMappedResizableDirectMemory(address, memoryReservationStep),
88     ↪ memoryReservationStep) { }
89
90     public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory)
91     ↪ : this(memory, DefaultLinksSizeStep) { }
92
93     public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory,
94     ↪ long memoryReservationStep)
95     {
96         Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
97         _memory = memory;
98         _memoryReservationStep = memoryReservationStep;
99         if (memory.ReservedCapacity < memoryReservationStep)
100         {
101             memory.ReservedCapacity = memoryReservationStep;
102         }
103         SetPointers(_memory);
104         // Гарантия корректности _memory.UsedCapacity относительно
105         ↪ _header->AllocatedLinks
106         _memory.UsedCapacity = ((long)_header->AllocatedLinks *
107         ↪ sizeof(Link)) + sizeof(LinksHeader);
108         // Гарантия корректности _header->ReservedLinks относительно
109         ↪ _memory.ReservedCapacity
110         _header->ReservedLinks = (id)((_memory.ReservedCapacity -
111         ↪ sizeof(LinksHeader)) / sizeof(Link));
112     }
113
114     [MethodImpl(MethodImplOptions.AggressiveInlining)]
115     public id Count(IList<id> restrictions)
116     {
117         // Если нет ограничений, тогда возвращаем общее число связей
118         ↪ находящихся в хранилище.
119         if (restrictions.Count == 0)
120         {
121             return Total;
122         }
123         if (restrictions.Count == 1)
124         {
125             var index = restrictions[Constants.IndexPart];
126             if (index == Constants.Any)
127             {
128                 return Total;
129             }
130             return Exists(index) ? 1UL : 0UL;
131         }
132         if (restrictions.Count == 2)
133         {
134             var index = restrictions[Constants.IndexPart];
135             var value = restrictions[1];
136             if (index == Constants.Any)
137             {
138                 if (value == Constants.Any)
139                 {
140                     return Total; // Any - как отсутствие ограничения
141                 }
142                 return _sourcesTreeMethods.CalculateReferences(value)
143                     + _targetsTreeMethods.CalculateReferences(value);
144             }
145             else

```



```

135 {
136     if (!Exists(index))
137     {
138         return 0;
139     }
140     if (value == Constants.Any)
141     {
142         return 1;
143     }
144     var storedLinkValue = GetLinkUnsafe(index);
145     if (storedLinkValue->Source == value ||
146         storedLinkValue->Target == value)
147     {
148         return 1;
149     }
150     return 0;
151 }
152 }
153 if (restrictions.Count == 3)
154 {
155     var index = restrictions[Constants.IndexPart];
156     var source = restrictions[Constants.SourcePart];
157     var target = restrictions[Constants.TargetPart];
158     if (index == Constants.Any)
159     {
160         if (source == Constants.Any && target == Constants.Any)
161         {
162             return Total;
163         }
164         else if (source == Constants.Any)
165         {
166             return _targetsTreeMethods.CalculateReferences(target);
167         }
168         else if (target == Constants.Any)
169         {
170             return _sourcesTreeMethods.CalculateReferences(source);
171         }
172         else //if(source != Any && target != Any)
173         {
174             // Эквивалент Exists(source, target) => Count(Any,
175             //     ↪ source, target) > 0
176             var link = _sourcesTreeMethods.Search(source, target);
177             return link == Constants.Null ? 0UL : 1UL;
178         }
179     }
180     else
181     {
182         if (!Exists(index))
183         {
184             return 0;
185         }
186         if (source == Constants.Any && target == Constants.Any)
187         {
188             return 1;
189         }
190         var storedLinkValue = GetLinkUnsafe(index);
191         if (source != Constants.Any && target != Constants.Any)
192         {
193             if (storedLinkValue->Source == source &&
194                 storedLinkValue->Target == target)
195             {
196                 return 1;
197             }
198             return 0;
199         }
200         var value = default(id);

```

```

200         if (source == Constants.Any)
201         {
202             value = target;
203         }
204         if (target == Constants.Any)
205         {
206             value = source;
207         }
208         if (storedLinkValue->Source == value ||
209             storedLinkValue->Target == value)
210         {
211             return 1;
212         }
213         return 0;
214     }
215 }
216 throw new NotSupportedException("Другие размеры и способы
    ↪ ограничений не поддерживаются.");
217 }
218
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
221 {
222     if (restrictions.Count == 0)
223     {
224         for (id link = 1; link <= _header->AllocatedLinks; link++)
225         {
226             if (Exists(link))
227             {
228                 if (handler(GetLinkStruct(link)) == Constants.Break)
229                 {
230                     return Constants.Break;
231                 }
232             }
233         }
234         return Constants.Continue;
235     }
236     if (restrictions.Count == 1)
237     {
238         var index = restrictions[Constants.IndexPart];
239         if (index == Constants.Any)
240         {
241             return Each(handler, ArrayPool<ulong>.Empty);
242         }
243         if (!Exists(index))
244         {
245             return Constants.Continue;
246         }
247         return handler(GetLinkStruct(index));
248     }
249     if (restrictions.Count == 2)
250     {
251         var index = restrictions[Constants.IndexPart];
252         var value = restrictions[1];
253         if (index == Constants.Any)
254         {
255             if (value == Constants.Any)
256             {
257                 return Each(handler, ArrayPool<ulong>.Empty);
258             }
259             if (Each(handler, new[] { index, value, Constants.Any })
260                 ↪ == Constants.Break)
261             {
262                 return Constants.Break;
263             }

```

```

263         return Each(handler, new[] { index, Constants.Any, value
264             ↪ });
265     }
266     else
267     {
268         if (!Exists(index))
269         {
270             return Constants.Continue;
271         }
272         if (value == Constants.Any)
273         {
274             return handler(GetLinkStruct(index));
275         }
276         var storedLinkValue = GetLinkUnsafe(index);
277         if (storedLinkValue->Source == value ||
278             storedLinkValue->Target == value)
279         {
280             return handler(GetLinkStruct(index));
281         }
282         return Constants.Continue;
283     }
284 }
285 if (restrictions.Count == 3)
286 {
287     var index = restrictions[Constants.IndexPart];
288     var source = restrictions[Constants.SourcePart];
289     var target = restrictions[Constants.TargetPart];
290     if (index == Constants.Any)
291     {
292         if (source == Constants.Any && target == Constants.Any)
293         {
294             return Each(handler, ArrayPool<ulong>.Empty);
295         }
296         else if (source == Constants.Any)
297         {
298             return _targetsTreeMethods.EachReference(target,
299                 ↪ handler);
300         }
301         else if (target == Constants.Any)
302         {
303             return _sourcesTreeMethods.EachReference(source,
304                 ↪ handler);
305         }
306         else //if(source != Any && target != Any)
307         {
308             var link = _sourcesTreeMethods.Search(source, target);
309             return link == Constants.Null ? Constants.Continue :
310                 ↪ handler(GetLinkStruct(link));
311         }
312     }
313 }
314 else
315 {
316     if (!Exists(index))
317     {
318         return Constants.Continue;
319     }
320     if (source == Constants.Any && target == Constants.Any)
321     {
322         return handler(GetLinkStruct(index));
323     }
324     var storedLinkValue = GetLinkUnsafe(index);
325     if (source != Constants.Any && target != Constants.Any)
326     {
327         if (storedLinkValue->Source == source &&
328             storedLinkValue->Target == target)

```

```

324         {
325             return handler(GetLinkStruct(index));
326         }
327         return Constants.Continue;
328     }
329     var value = default(id);
330     if (source == Constants.Any)
331     {
332         value = target;
333     }
334     if (target == Constants.Any)
335     {
336         value = source;
337     }
338     if (storedLinkValue->Source == value ||
339         storedLinkValue->Target == value)
340     {
341         return handler(GetLinkStruct(index));
342     }
343     return Constants.Continue;
344 }
345 }
346 throw new NotSupportedException("Другие размеры и способы
347     ↪ ограничений не поддерживаются.");
348 }
349
350 /// <remarks>
351 /// TODO: Возможно можно перемещать значения, если указан индекс, но
352 /// значение существует в другом месте (но не в менеджере памяти, а в
353 /// логике Links)
354 /// </remarks>
355 [MethodImpl(MethodImplOptions.AggressiveInlining)]
356 public id Update(IList<id> values)
357 {
358     var linkIndex = values[Constants.IndexPart];
359     var link = GetLinkUnsafe(linkIndex);
360     // Будет корректно работать только в том случае, если пространство
361     ↪ выделенной связи предварительно заполнено нулями
362     if (link->Source != Constants.Null)
363     {
364         _sourcesTreeMethods.Detach(new
365             ↪ IntPtr(&_header->FirstAsSource), linkIndex);
366     }
367     if (link->Target != Constants.Null)
368     {
369         _targetsTreeMethods.Detach(new
370             ↪ IntPtr(&_header->FirstAsTarget), linkIndex);
371     }
372 }
373 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
374     var leftTreeSize = _sourcesTreeMethods.GetSize(new
375         ↪ IntPtr(&_header->FirstAsSource));
376     var rightTreeSize = _targetsTreeMethods.GetSize(new
377         ↪ IntPtr(&_header->FirstAsTarget));
378     if (leftTreeSize != rightTreeSize)
379     {
380         throw new Exception("One of the trees is broken.");
381     }
382 }
383 #endif
384 link->Source = values[Constants.SourcePart];
385 link->Target = values[Constants.TargetPart];
386 if (link->Source != Constants.Null)
387 {
388     _sourcesTreeMethods.Attach(new
389         ↪ IntPtr(&_header->FirstAsSource), linkIndex);
390 }

```

```

380         if (link->Target != Constants.Null)
381         {
382             _targetsTreeMethods.Attach(new
383                 ↳ IntPtr(&_header->FirstAsTarget), linkIndex);
384 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
385             leftTreeSize = _sourcesTreeMethods.GetSize(new
386                 ↳ IntPtr(&_header->FirstAsSource));
387             rightTreeSize = _targetsTreeMethods.GetSize(new
388                 ↳ IntPtr(&_header->FirstAsTarget));
389             if (leftTreeSize != rightTreeSize)
390             {
391                 throw new Exception("One of the trees is broken.");
392             }
393 #endif
394             return linkIndex;
395         }
396 [MethodImpl(MethodImplOptions.AggressiveInlining)]
397 private IList<id> GetLinkStruct(id linkIndex)
398 {
399     var link = GetLinkUnsafe(linkIndex);
400     return new UInt64Link(linkIndex, link->Source, link->Target);
401 }
402 [MethodImpl(MethodImplOptions.AggressiveInlining)]
403 private Link* GetLinkUnsafe(id linkIndex) => &_amp;links[linkIndex];
404
405 /// <remarks>
406 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими
407 ↳ не заполняет пространство
408 /// </remarks>
409 public id Create()
410 {
411     var freeLink = _header->FirstFreeLink;
412     if (freeLink != Constants.Null)
413     {
414         _unusedLinksListMethods.Detach(freeLink);
415     }
416     else
417     {
418         if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
419         {
420             throw new
421                 ↳ LinksLimitReachedException(Constants.MaxPossibleIndex);
422         }
423         if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
424         {
425             _memory.ReservedCapacity += _memoryReservationStep;
426             SetPointers(_memory);
427             _header->ReservedLinks = (id)(_memory.ReservedCapacity /
428                 ↳ sizeof(Link));
429         }
430         _header->AllocatedLinks++;
431         _memory.UsedCapacity += sizeof(Link);
432         freeLink = _header->AllocatedLinks;
433     }
434     return freeLink;
435 }
436
437 public void Delete(id link)
438 {
439     if (link < _header->AllocatedLinks)
440     {
441         _unusedLinksListMethods.AttachAsFirst(link);
442     }
443 }

```

```

440     else if (link == _header->AllocatedLinks)
441     {
442         _header->AllocatedLinks--;
443         _memory.UsedCapacity -= sizeof(Link);
444         // Убираем все связи, находящиеся в списке свободных в конце
445         ↳ файла, до тех пор, пока не дойдём до первой существующей
446         ↳ связи
447         // Позволяет оптимизировать количество выделенных связей
448         ↳ (AllocatedLinks)
449         while (_header->AllocatedLinks > 0 &&
450             ↳ IsUnusedLink(_header->AllocatedLinks))
451         {
452             _unusedLinksListMethods.Detach(_header->AllocatedLinks);
453             _header->AllocatedLinks--;
454             _memory.UsedCapacity -= sizeof(Link);
455         }
456     }
457 }
458
459 /// <remarks>
460 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в
461 ↳ том случае, если адрес реально поменялся
462 ///
463 /// Указатель this.links может быть в том же месте,
464 ↳ так как 0-я связь не используется и имеет такой же размер как
465 ↳ Header,
466 ↳ поэтому header размещается в том же месте, что и 0-я связь
467 /// </remarks>
468 private void SetPointers(IResizableDirectMemory memory)
469 {
470     if (memory == null)
471     {
472         _header = null;
473         _links = null;
474         _unusedLinksListMethods = null;
475         _targetsTreeMethods = null;
476         _unusedLinksListMethods = null;
477     }
478     else
479     {
480         _header = (LinksHeader*)(void*)memory.Pointer;
481         _links = (Link*)(void*)memory.Pointer;
482         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
483         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
484         _unusedLinksListMethods = new UnusedLinksListMethods(_links,
485             ↳ _header);
486     }
487 }
488
489 [MethodImpl(MethodImplOptions.AggressiveInlining)]
490 private bool Exists(id link) => link >= Constants.MinPossibleIndex &&
491     ↳ link <= _header->AllocatedLinks && !IsUnusedLink(link);
492
493 [MethodImpl(MethodImplOptions.AggressiveInlining)]
494 private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
495     ↳ || (_links[link].SizeAsSource ==
496         ↳ Constants.Null &&
497         ↳ _links[link].Source !=
498         ↳ Constants.Null);
499
500 #region Disposable
501 protected override bool AllowMultipleDisposeCalls => true;
502
503 protected override void DisposeCore(bool manual, bool wasDisposed)
504 {

```

```

495         if (!wasDisposed)
496         {
497             SetPointers(null);
498         }
499         Disposable.TryDispose(_memory);
500     }
501
502     #endregion
503 }
504 }

```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs

```

1  using Platform.Collections.Methods.Lists;
2
3  namespace Platform.Data.Doublets.ResizableDirectMemory
4  {
5      unsafe partial class UInt64ResizableDirectMemoryLinks
6      {
7          private class UnusedLinksListMethods :
8              ↳ CircularDoublyLinkedListMethods<ulong>
9          {
10             private readonly Link* _links;
11             private readonly LinksHeader* _header;
12
13             public UnusedLinksListMethods(Link* links, LinksHeader* header)
14             {
15                 _links = links;
16                 _header = header;
17             }
18
19             protected override ulong GetFirst() => _header->FirstFreeLink;
20
21             protected override ulong GetLast() => _header->LastFreeLink;
22
23             protected override ulong GetPrevious(ulong element) =>
24                 ↳ _links[element].Source;
25
26             protected override ulong GetNext(ulong element) =>
27                 ↳ _links[element].Target;
28
29             protected override ulong GetSize() => _header->FreeLinks;
30
31             protected override void SetFirst(ulong element) =>
32                 ↳ _header->FirstFreeLink = element;
33
34             protected override void SetLast(ulong element) =>
35                 ↳ _header->LastFreeLink = element;
36
37             protected override void SetPrevious(ulong element, ulong previous)
38                 ↳ => _links[element].Source = previous;
39
40             protected override void SetNext(ulong element, ulong next) =>
41                 ↳ _links[element].Target = next;
42
43             protected override void SetSize(ulong size) => _header->FreeLinks
44                 ↳ = size;
45         }
46     }
47 }

```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs

```

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

```

```

7
8  namespace Platform.Data.Doublets.ResizableDirectMemory
9  {
10     unsafe partial class UInt64ResizableDirectMemoryLinks
11     {
12         private abstract class LinksTreeMethodsBase :
13             ↳ SizedAndThreadedAVLBalancedTreeMethods<ulong>
14         {
15             private readonly UInt64ResizableDirectMemoryLinks _memory;
16             private readonly LinksCombinedConstants<ulong, ulong, int>
17                 ↳ _constants;
18             protected readonly Link* Links;
19             protected readonly LinksHeader* Header;
20
21             protected LinksTreeMethodsBase(UInt64ResizableDirectMemoryLinks
22                 ↳ memory)
23             {
24                 Links = memory._links;
25                 Header = memory._header;
26                 _memory = memory;
27                 _constants = memory.Constants;
28             }
29
30             [MethodImpl(MethodImplOptions.AggressiveInlining)]
31             protected abstract ulong GetTreeRoot();
32
33             [MethodImpl(MethodImplOptions.AggressiveInlining)]
34             protected abstract ulong GetBasePartValue(ulong link);
35
36             public ulong this[ulong index]
37             {
38                 get
39                 {
40                     var root = GetTreeRoot();
41                     if (index >= GetSize(root))
42                     {
43                         return 0;
44                     }
45                     while (root != 0)
46                     {
47                         var left = GetLeftOrDefault(root);
48                         var leftSize = GetSizeOrZero(left);
49                         if (index < leftSize)
50                         {
51                             root = left;
52                             continue;
53                         }
54                         if (index == leftSize)
55                         {
56                             return root;
57                         }
58                         root = GetRightOrDefault(root);
59                         index -= leftSize + 1;
60                     }
61                     return 0; // TODO: Impossible situation exception (only if
62                         ↳ tree structure broken)
63                 }
64             }
65
66             // TODO: Return indices range instead of references count
67             public ulong CalculateReferences(ulong link)
68             {
69                 var root = GetTreeRoot();
70                 var total = GetSize(root);
71                 var totalRightIgnore = OUL;
72                 while (root != 0)

```

```

69     {
70         var @base = GetBasePartValue(root);
71         if (@base <= link)
72         {
73             root = GetRightOrDefault(root);
74         }
75         else
76         {
77             totalRightIgnore += GetRightSize(root) + 1;
78             root = GetLeftOrDefault(root);
79         }
80     }
81     root = GetTreeRoot();
82     var totalLeftIgnore = 0UL;
83     while (root != 0)
84     {
85         var @base = GetBasePartValue(root);
86         if (@base >= link)
87         {
88             root = GetLeftOrDefault(root);
89         }
90         else
91         {
92             totalLeftIgnore += GetLeftSize(root) + 1;
93             root = GetRightOrDefault(root);
94         }
95     }
96     return total - totalRightIgnore - totalLeftIgnore;
97 }
98
99 public ulong EachReference(ulong link, Func<IList<ulong>, ulong>
↳ handler)
100 {
101     var root = GetTreeRoot();
102     if (root == 0)
103     {
104         return _constants.Continue;
105     }
106     ulong first = 0, current = root;
107     while (current != 0)
108     {
109         var @base = GetBasePartValue(current);
110         if (@base >= link)
111         {
112             if (@base == link)
113             {
114                 first = current;
115             }
116             current = GetLeftOrDefault(current);
117         }
118         else
119         {
120             current = GetRightOrDefault(current);
121         }
122     }
123     if (first != 0)
124     {
125         current = first;
126         while (true)
127         {
128             if (handler(_memory.GetLinkStruct(current)) ==
↳ _constants.Break)
129             {
130                 return _constants.Break;
131             }
132             current = GetNext(current);

```

```

133         if (current == 0 || GetBasePartValue(current) != link)
134         {
135             break;
136         }
137     }
138     }
139     return _constants.Continue;
140 }
141
142 protected override void PrintNodeValue(ulong node, StringBuilder
↳ sb)
143 {
144     sb.Append(' ');
145     sb.Append(Links[node].Source);
146     sb.Append('-');
147     sb.Append('>');
148     sb.Append(Links[node].Target);
149 }
150
151 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
152 {
153     public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks
↳ memory)
154     : base(memory)
155     {
156     }
157
158     protected override IntPtr GetLeftPointer(ulong node) => new
↳ IntPtr(&Links[node].LeftAsSource);
159
160     protected override IntPtr GetRightPointer(ulong node) => new
↳ IntPtr(&Links[node].RightAsSource);
161
162     protected override ulong GetLeftValue(ulong node) =>
↳ Links[node].LeftAsSource;
163
164     protected override ulong GetRightValue(ulong node) =>
↳ Links[node].RightAsSource;
165
166     protected override ulong GetSize(ulong node)
167     {
168         var previousValue = Links[node].SizeAsSource;
169         //return MathHelpers.PartialRead(previousValue, 5, -5);
170         return (previousValue & 4294967264) >> 5;
171     }
172
173     protected override void SetLeft(ulong node, ulong left) =>
↳ Links[node].LeftAsSource = left;
174
175     protected override void SetRight(ulong node, ulong right) =>
↳ Links[node].RightAsSource = right;
176
177     protected override void SetSize(ulong node, ulong size)
178     {
179         var previousValue = Links[node].SizeAsSource;
180         //var modified = MathHelpers.PartialWrite(previousValue, size,
↳ 5, -5);
181         var modified = (previousValue & 31) | ((size & 134217727) <<
↳ 5);
182         Links[node].SizeAsSource = modified;
183     }
184
185     protected override bool GetLeftIsChild(ulong node)
186     {
187

```

```

188     var previousValue = Links[node].SizeAsSource;
189     //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
190     return (previousValue & 16) >> 4 == 1UL;
191 }
192
193 protected override void SetLeftIsChild(ulong node, bool value)
194 {
195     var previousValue = Links[node].SizeAsSource;
196     //var modified = MathHelpers.PartialWrite(previousValue,
197     ↪ (ulong)(Integer)value, 4, 1);
198     var modified = (previousValue & 4294967279) | ((value ? 1UL :
199     ↪ 0UL) << 4);
200     Links[node].SizeAsSource = modified;
201 }
202
203 protected override bool GetRightIsChild(ulong node)
204 {
205     var previousValue = Links[node].SizeAsSource;
206     //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
207     return (previousValue & 8) >> 3 == 1UL;
208 }
209
210 protected override void SetRightIsChild(ulong node, bool value)
211 {
212     var previousValue = Links[node].SizeAsSource;
213     //var modified = MathHelpers.PartialWrite(previousValue,
214     ↪ (ulong)(Integer)value, 3, 1);
215     var modified = (previousValue & 4294967287) | ((value ? 1UL :
216     ↪ 0UL) << 3);
217     Links[node].SizeAsSource = modified;
218 }
219
220 protected override sbyte GetBalance(ulong node)
221 {
222     var previousValue = Links[node].SizeAsSource;
223     //var value = MathHelpers.PartialRead(previousValue, 0, 3);
224     var value = previousValue & 7;
225     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) <<
226     ↪ 5) | value & 3 | 124 : value & 3);
227     return unpackedValue;
228 }
229
230 protected override void SetBalance(ulong node, sbyte value)
231 {
232     var previousValue = Links[node].SizeAsSource;
233     var packagedValue = (ulong)((((byte)value >> 5) & 4) | value &
234     ↪ 3);
235     //var modified = MathHelpers.PartialWrite(previousValue,
236     ↪ packagedValue, 0, 3);
237     var modified = (previousValue & 4294967288) | (packagedValue &
238     ↪ 7);
239     Links[node].SizeAsSource = modified;
240 }
241
242 protected override bool FirstIsToTheLeftOfSecond(ulong first,
243     ↪ ulong second)
244     => Links[first].Source < Links[second].Source ||
245     (Links[first].Source == Links[second].Source &&
246     ↪ Links[first].Target < Links[second].Target);
247
248 protected override bool FirstIsToTheRightOfSecond(ulong first,
249     ↪ ulong second)
250     => Links[first].Source > Links[second].Source ||
251     (Links[first].Source == Links[second].Source &&
252     ↪ Links[first].Target > Links[second].Target);

```

```

241 protected override ulong GetTreeRoot() => Header->FirstAsSource;
242
243 protected override ulong GetBasePartValue(ulong link) =>
244     ↪ Links[link].Source;
245
246 /// <summary>
247 /// Выполняет поиск и возвращает индекс связи с указанными Source
248     ↪ (началом) и Target (концом)
249 /// по дереву (индексу) связей, отсортированному по Source, а
250     ↪ затем по Target.
251 /// </summary>
252 /// <param name="source">Индекс связи, которая является началом на
253     ↪ искомой связи.</param>
254 /// <param name="target">Индекс связи, которая является концом на
255     ↪ искомой связи.</param>
256 /// <returns>Индекс искомой связи.</returns>
257 public ulong Search(ulong source, ulong target)
258 {
259     var root = Header->FirstAsSource;
260     while (root != 0)
261     {
262         var rootSource = Links[root].Source;
263         var rootTarget = Links[root].Target;
264         if (FirstIsToTheLeftOfSecond(source, target, rootSource,
265         ↪ rootTarget)) // node.Key < root.Key
266         {
267             root = GetLeftOrDefault(root);
268         }
269         else if (FirstIsToTheRightOfSecond(source, target,
270         ↪ rootSource, rootTarget)) // node.Key > root.Key
271         {
272             root = GetRightOrDefault(root);
273         }
274         else // node.Key == root.Key
275         {
276             return root;
277         }
278     }
279     return 0;
280 }
281
282 [MethodImpl(MethodImplOptions.AggressiveInlining)]
283 private static bool FirstIsToTheLeftOfSecond(ulong firstSource,
284     ↪ ulong firstTarget, ulong secondSource, ulong secondTarget)
285     => firstSource < secondSource || (firstSource == secondSource
286     ↪ && firstTarget < secondTarget);
287
288 [MethodImpl(MethodImplOptions.AggressiveInlining)]
289 private static bool FirstIsToTheRightOfSecond(ulong firstSource,
290     ↪ ulong firstTarget, ulong secondSource, ulong secondTarget)
291     => firstSource > secondSource || (firstSource == secondSource
292     ↪ && firstTarget > secondTarget);
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 protected override void ClearNode(ulong node)
296 {
297     Links[node].LeftAsSource = 0UL;
298     Links[node].RightAsSource = 0UL;
299     Links[node].SizeAsSource = 0UL;
300 }
301
302 [MethodImpl(MethodImplOptions.AggressiveInlining)]
303 protected override ulong GetZero() => 0UL;

```

```

294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 protected override ulong GetOne() => 1UL;
296
297 [MethodImpl(MethodImplOptions.AggressiveInlining)]
298 protected override ulong GetTwo() => 2UL;
299
300 [MethodImpl(MethodImplOptions.AggressiveInlining)]
301 protected override bool ValueEqualToZero(IntPtr pointer) =>
302     ↳ *(ulong*)pointer.ToPointer() == 0UL;
303
304 [MethodImpl(MethodImplOptions.AggressiveInlining)]
305 protected override bool EqualToZero(ulong value) => value == 0UL;
306
307 [MethodImpl(MethodImplOptions.AggressiveInlining)]
308 protected override bool IsEquals(ulong first, ulong second) =>
309     ↳ first == second;
310
311 [MethodImpl(MethodImplOptions.AggressiveInlining)]
312 protected override bool GreaterThanZero(ulong value) => value >
313     ↳ 0UL;
314
315 [MethodImpl(MethodImplOptions.AggressiveInlining)]
316 protected override bool GreaterThan(ulong first, ulong second) =>
317     ↳ first > second;
318
319 [MethodImpl(MethodImplOptions.AggressiveInlining)]
320 protected override bool GreaterOrEqualThan(ulong first, ulong
321     ↳ second) => first >= second;
322
323 [MethodImpl(MethodImplOptions.AggressiveInlining)]
324 protected override bool GreaterOrEqualThanZero(ulong value) =>
325     ↳ true; // value >= 0 is always true for ulong
326
327 [MethodImpl(MethodImplOptions.AggressiveInlining)]
328 protected override bool LessOrEqualThanZero(ulong value) => value
329     ↳ == 0; // value is always >= 0 for ulong
330
331 [MethodImpl(MethodImplOptions.AggressiveInlining)]
332 protected override bool LessOrEqualThan(ulong first, ulong second)
333     ↳ => first <= second;
334
335 [MethodImpl(MethodImplOptions.AggressiveInlining)]
336 protected override bool LessThanZero(ulong value) => false; //
337     ↳ value < 0 is always false for ulong
338
339 [MethodImpl(MethodImplOptions.AggressiveInlining)]
340 protected override bool LessThan(ulong first, ulong second) =>
341     ↳ first < second;
342
343 [MethodImpl(MethodImplOptions.AggressiveInlining)]
344 protected override bool LessThan(ulong first, ulong second) =>
345     ↳ first < second;
346
347 private class LinksTargetsTreeMethods : LinksTreeMethodsBase

```

```

348 {
349     public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks
350         ↳ memory)
351         : base(memory)
352     {
353     }
354
355     //protected override IntPtr GetLeft(ulong node) => new
356     ↳ IntPtr(&Links[node].LeftAsTarget);
357
358     //protected override IntPtr GetRight(ulong node) => new
359     ↳ IntPtr(&Links[node].RightAsTarget);
360
361     //protected override ulong GetSize(ulong node) =>
362     ↳ Links[node].SizeAsTarget;
363
364     //protected override void SetLeft(ulong node, ulong left) =>
365     ↳ Links[node].LeftAsTarget = left;
366
367     //protected override void SetRight(ulong node, ulong right) =>
368     ↳ Links[node].RightAsTarget = right;
369
370     //protected override void SetSize(ulong node, ulong size) =>
371     ↳ Links[node].SizeAsTarget = size;
372
373     protected override IntPtr GetLeftPointer(ulong node) => new
374     ↳ IntPtr(&Links[node].LeftAsTarget);
375
376     protected override IntPtr GetRightPointer(ulong node) => new
377     ↳ IntPtr(&Links[node].RightAsTarget);
378
379     protected override ulong GetLeftValue(ulong node) =>
380     ↳ Links[node].LeftAsTarget;
381
382     protected override ulong GetRightValue(ulong node) =>
383     ↳ Links[node].RightAsTarget;
384
385     protected override ulong GetSize(ulong node)
386     {
387         var previousValue = Links[node].SizeAsTarget;
388         //return MathHelpers.PartialRead(previousValue, 5, -5);
389         return (previousValue & 4294967264) >> 5;
390     }
391
392     protected override void SetLeft(ulong node, ulong left) =>
393     ↳ Links[node].LeftAsTarget = left;
394
395     protected override void SetRight(ulong node, ulong right) =>
396     ↳ Links[node].RightAsTarget = right;
397
398     protected override void SetSize(ulong node, ulong size)
399     {
400         var previousValue = Links[node].SizeAsTarget;
401         //var modified = MathHelpers.PartialWrite(previousValue, size,
402         ↳ 5, -5);
403         var modified = (previousValue & 31) | ((size & 134217727) <<
404         ↳ 5);
405         Links[node].SizeAsTarget = modified;
406     }
407
408     protected override bool GetLeftIsChild(ulong node)
409     {
410         var previousValue = Links[node].SizeAsTarget;
411         //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
412         return (previousValue & 16) >> 4 == 1UL;
413     }
414 }

```

```

398     // TODO: Check if this is possible to use
399     //var nodeSize = GetSize(node);
400     //var left = GetLeftValue(node);
401     //var leftSize = GetSizeOrZero(left);
402     //return leftSize > 0 && nodeSize > leftSize;
403 }
404
405 protected override void SetLeftIsChild(ulong node, bool value)
406 {
407     var previousValue = Links[node].SizeAsTarget;
408     //var modified = MathHelpers.PartialWrite(previousValue,
409     ↪ (ulong)(Integer)value, 4, 1);
410     var modified = (previousValue & 4294967279) | ((value ? 1UL :
411     ↪ 0UL) << 4);
412     Links[node].SizeAsTarget = modified;
413 }
414
415 protected override bool GetRightIsChild(ulong node)
416 {
417     var previousValue = Links[node].SizeAsTarget;
418     //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
419     return (previousValue & 8) >> 3 == 1UL;
420     // TODO: Check if this is possible to use
421     //var nodeSize = GetSize(node);
422     //var right = GetRightValue(node);
423     //var rightSize = GetSizeOrZero(right);
424     //return rightSize > 0 && nodeSize > rightSize;
425 }
426
427 protected override void SetRightIsChild(ulong node, bool value)
428 {
429     var previousValue = Links[node].SizeAsTarget;
430     //var modified = MathHelpers.PartialWrite(previousValue,
431     ↪ (ulong)(Integer)value, 3, 1);
432     var modified = (previousValue & 4294967287) | ((value ? 1UL :
433     ↪ 0UL) << 3);
434     Links[node].SizeAsTarget = modified;
435 }
436
437 protected override sbyte GetBalance(ulong node)
438 {
439     var previousValue = Links[node].SizeAsTarget;
440     //var value = MathHelpers.PartialRead(previousValue, 0, 3);
441     var value = previousValue & 7;
442     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) <<
443     ↪ 5) | value & 3 | 124 : value & 3);
444     return unpackedValue;
445 }
446
447 protected override void SetBalance(ulong node, sbyte value)
448 {
449     var previousValue = Links[node].SizeAsTarget;
450     var packagedValue = (ulong)((((byte)value >> 5) & 4) | value &
451     ↪ 3);
452     //var modified = MathHelpers.PartialWrite(previousValue,
453     ↪ packagedValue, 0, 3);
454     var modified = (previousValue & 4294967288) | (packagedValue &
455     ↪ 7);
456     Links[node].SizeAsTarget = modified;
457 }
458
459 protected override bool FirstIsToTheLeftOfSecond(ulong first,
460 ↪ ulong second)
461     => Links[first].Target < Links[second].Target ||

```

```

453     (Links[first].Target == Links[second].Target &&
454     ↪ Links[first].Source < Links[second].Source);
455
456 protected override bool FirstIsToTheRightOfSecond(ulong first,
457 ↪ ulong second)
458     => Links[first].Target > Links[second].Target ||
459     (Links[first].Target == Links[second].Target &&
460     ↪ Links[first].Source > Links[second].Source);
461
462 protected override ulong GetTreeRoot() => Header->FirstAsTarget;
463
464 protected override ulong GetBasePartValue(ulong link) =>
465     ↪ Links[link].Target;
466
467 [MethodImpl(MethodImplOptions.AggressiveInlining)]
468 protected override void ClearNode(ulong node)
469 {
470     Links[node].LeftAsTarget = 0UL;
471     Links[node].RightAsTarget = 0UL;
472     Links[node].SizeAsTarget = 0UL;
473 }
474 }
475 }
476 }
477 }
478 }
479 }
480 }
481 }
482 }

```

./Sequences/Converters/BalancedVariantConverter.cs

```

1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Sequences.Converters
4 {
5     public class BalancedVariantConverter<TLink> :
6     ↪ LinksListToSequenceConverterBase<TLink>
7     {
8         public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
9
10        public override TLink Convert(ICollection<TLink> sequence)
11        {
12            var length = sequence.Count;
13            if (length < 1)
14            {
15                return default;
16            }
17            if (length == 1)
18            {
19                return sequence[0];
20            }
21            // Make copy of next layer
22            if (length > 2)
23            {
24                // TODO: Try to use stackalloc (which at the moment is not
25                ↪ working with generics) but will be possible with Sigil
26                var halvedSequence = new TLink[(length / 2) + (length % 2)];
27                HalveSequence(halvedSequence, sequence, length);
28                sequence = halvedSequence;
29                length = halvedSequence.Length;
30            }
31            // Keep creating layer after layer
32            while (length > 2)
33            {
34                HalveSequence(sequence, sequence, length);
35                length = (length / 2) + (length % 2);
36            }
37            return Links.GetOrCreate(sequence[0], sequence[1]);
38        }
39    }
40 }

```



```

38     private void HalveSequence(IList<TLink> destination, IList<TLink>
    ↪ source, int length)
39     {
40         var loopedLength = length - (length % 2);
41         for (var i = 0; i < loopedLength; i += 2)
42         {
43             destination[i / 2] = Links.GetOrCreate(source[i], source[i +
    ↪ 1]);
44         }
45         if (length > loopedLength)
46         {
47             destination[length / 2] = source[length - 1];
48         }
49     }
50 }
51 }

```

./Sequences/Converters/CompressingConverter.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Collections;
6  using Platform.Helpers.Singletons;
7  using Platform.Numbers;
8  using Platform.Data.Constants;
9  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
10
11 namespace Platform.Data.Doublets.Sequences.Converters
12 {
13     /// <remarks>
14     /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью
    ↪ изолированно от Links на этапе сжатия.
15     /// А именно будет создаваться временный список пар необходимых для
    ↪ выполнения сжатия, в таком случае тип значения элемента массива может
    ↪ быть любым, как char так и ulong.
16     /// Как только список/словарь пар был выявлен можно разом выполнить
    ↪ создание всех этих пар, а так же разом выполнить замену.
17     /// </remarks>
18     public class CompressingConverter<TLink> :
    ↪ LinksListToSequenceConverterBase<TLink>
19     {
20         private static readonly LinksCombinedConstants<bool, TLink, long>
    ↪ _constants = Default<LinksCombinedConstants<bool, TLink,
    ↪ long>>.Instance;
21         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪ EqualityComparer<TLink>.Default;
22         private static readonly Comparer<TLink> _comparer =
    ↪ Comparer<TLink>.Default;
23
24         private readonly IConverter<IList<TLink>, TLink> _baseConverter;
25         private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
26         private readonly TLink _minFrequencyToCompress;
27         private readonly bool _doInitialFrequenciesIncrement;
28         private Doublet<TLink> _maxDoublet;
29         private LinkFrequency<TLink> _maxDoubletData;
30
31         private struct HalfDoublet
32         {
33             public TLink Element;
34             public LinkFrequency<TLink> DoubletData;
35
36             public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
37             {
38                 Element = element;
39                 DoubletData = doubletData;
40             }
41         }

```

```

42         public override string ToString() => $"{Element}: ({DoubletData})";
43     }
44
45     public CompressingConverter(ILinks<TLink> links,
    ↪ IConverter<IList<TLink>, TLink> baseConverter,
    ↪ LinkFrequenciesCache<TLink> doubletFrequenciesCache)
46     : this(links, baseConverter, doubletFrequenciesCache,
    ↪ Integer<TLink>.One, true)
47     {
48     }
49
50     public CompressingConverter(ILinks<TLink> links,
    ↪ IConverter<IList<TLink>, TLink> baseConverter,
    ↪ LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
    ↪ doInitialFrequenciesIncrement)
51     : this(links, baseConverter, doubletFrequenciesCache,
    ↪ Integer<TLink>.One, doInitialFrequenciesIncrement)
52     {
53     }
54
55     public CompressingConverter(ILinks<TLink> links,
    ↪ IConverter<IList<TLink>, TLink> baseConverter,
    ↪ LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
    ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
56     : base(links)
57     {
58         _baseConverter = baseConverter;
59         _doubletFrequenciesCache = doubletFrequenciesCache;
60         if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One)
    ↪ < 0)
61         {
62             minFrequencyToCompress = Integer<TLink>.One;
63         }
64         _minFrequencyToCompress = minFrequencyToCompress;
65         _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
66         ResetMaxDoublet();
67     }
68
69     public override TLink Convert(IList<TLink> source) =>
    ↪ _baseConverter.Convert(Compress(source));
70
71     /// <remarks>
72     /// Original algorithm idea:
    ↪ https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
73     /// Faster version (doublets' frequencies dictionary is not recreated).
74     /// </remarks>
75     private IList<TLink> Compress(IList<TLink> sequence)
76     {
77         if (sequence.IsNullOrEmpty())
78         {
79             return null;
80         }
81         if (sequence.Count == 1)
82         {
83             return sequence;
84         }
85         if (sequence.Count == 2)
86         {
87             return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
88         }
89         // TODO: arraypool with min size (to improve cache locality) or
    ↪ stackallow with Sigil
90         var copy = new HalfDoublet[sequence.Count];
91         Doublet<TLink> doublet = default;
92         for (var i = 1; i < sequence.Count; i++)

```

```

93     {
94         doublet.Source = sequence[i - 1];
95         doublet.Target = sequence[i];
96         LinkFrequency<TLink> data;
97         if (_doInitialFrequenciesIncrement)
98         {
99             data = _doubletFrequenciesCache.IncrementFrequency(ref
100                 ↳ doublet);
101         }
102         else
103         {
104             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
105             if (data == null)
106             {
107                 throw new NotSupportedException("If you ask not to
108                 ↳ increment frequencies, it is expected that all
109                 ↳ frequencies for the sequence are prepared.");
110             }
111             copy[i - 1].Element = sequence[i - 1];
112             copy[i - 1].DoubletData = data;
113             UpdateMaxDoublet(ref doublet, data);
114         }
115         copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
116         copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
117         if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
118         {
119             var newLength = ReplaceDoublets(copy);
120             sequence = new TLink[newLength];
121             for (int i = 0; i < newLength; i++)
122             {
123                 sequence[i] = copy[i].Element;
124             }
125         }
126         return sequence;
127     }
128     /// <remarks>
129     /// Original algorithm idea:
130     ↳ https://en.wikipedia.org/wiki/Byte_pair_encoding
131     /// </remarks>
132     private int ReplaceDoublets(HalfDoublet[] copy)
133     {
134         var oldLength = copy.Length;
135         var newLength = copy.Length;
136         while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
137         {
138             var maxDoubletSource = _maxDoublet.Source;
139             var maxDoubletTarget = _maxDoublet.Target;
140             if (_equalityComparer.Equals(_maxDoubletData.Link,
141                 ↳ _constants.Null))
142             {
143                 _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource,
144                 ↳ maxDoubletTarget);
145             }
146             var maxDoubletReplacementLink = _maxDoubletData.Link;
147             oldLength--;
148             var oldLengthMinusTwo = oldLength - 1;
149             // Substitute all usages
150             int w = 0, r = 0; // (r == read, w == write)
151             for (; r < oldLength; r++)
152             {
153                 if (_equalityComparer.Equals(copy[r].Element,
154                 ↳ maxDoubletSource) && _equalityComparer.Equals(copy[r +
155                 ↳ 1].Element, maxDoubletTarget))

```

```

150     {
151         if (r > 0)
152         {
153             var previous = copy[w - 1].Element;
154             copy[w - 1].DoubletData.DecrementFrequency();
155             copy[w - 1].DoubletData = _doubletFrequenciesCache
156                 ↳ .IncrementFrequency(previous,
157                 ↳ maxDoubletReplacementLink);
158         }
159         if (r < oldLengthMinusTwo)
160         {
161             var next = copy[r + 2].Element;
162             copy[r + 1].DoubletData.DecrementFrequency();
163             copy[w].DoubletData = _doubletFrequenciesCache.Inc
164                 ↳ rementFrequency(maxDoubletReplacementLink,
165                 ↳ next);
166         }
167         copy[w++] .Element = maxDoubletReplacementLink;
168         r++;
169         newLength--;
170     }
171     else
172     {
173         copy[w++] = copy[r];
174     }
175     if (w < newLength)
176     {
177         copy[w] = copy[r];
178     }
179     oldLength = newLength;
180     ResetMaxDoublet();
181     UpdateMaxDoublet(copy, newLength);
182     }
183     return newLength;
184 }
185 [MethodImpl(MethodImplOptions.AggressiveInlining)]
186 private void ResetMaxDoublet()
187 {
188     _maxDoublet = new Doublet<TLink>();
189     _maxDoubletData = new LinkFrequency<TLink>();
190 }
191 [MethodImpl(MethodImplOptions.AggressiveInlining)]
192 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
193 {
194     Doublet<TLink> doublet = default;
195     for (var i = 1; i < length; i++)
196     {
197         doublet.Source = copy[i - 1].Element;
198         doublet.Target = copy[i].Element;
199         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
200     }
201 }
202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 private void UpdateMaxDoublet(ref Doublet<TLink> doublet,
204     ↳ LinkFrequency<TLink> data)
205 {
206     var frequency = data.Frequency;
207     var maxFrequency = _maxDoubletData.Frequency;

```

```

207 //if (frequency > _minFrequencyToCompress && (maxFrequency <
    frequency || (maxFrequency == frequency && doublet.Source +
    ↳ doublet.Target < /* gives better compression string data (and
    ↳ gives collisions quickly) */ _maxDoublet.Source +
    ↳ _maxDoublet.Target)))
208 if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
209     _comparer.Compare(maxFrequency, frequency) < 0 ||
    (_equalityComparer.Equals(maxFrequency, frequency) &&
    ↳ _comparer.Compare(ArithmeticHelpers.Add(doublet.Source,
    ↳ doublet.Target), ArithmeticHelpers.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) */
210 {
211     _maxDoublet = doublet;
212     _maxDoubletData = data;
213 }
214 }
215 }
216 }

```

./Sequences/Converters/LinksListToSequenceConverterBase.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Sequences.Converters
5 {
6     public abstract class LinksListToSequenceConverterBase<TLink> :
    ↳ IConverter<IList<TLink>, TLink>
7     {
8         protected readonly ILinks<TLink> Links;
9         public LinksListToSequenceConverterBase(ILinks<TLink> links) => Links
    ↳ = links;
10        public abstract TLink Convert(IList<TLink> source);
11    }
12 }

```

./Sequences/Converters/OptimalVariantConverter.cs

```

1 using System.Collections.Generic;
2 using System.Linq;
3 using Platform.Interfaces;
4
5 namespace Platform.Data.Doublets.Sequences.Converters
6 {
7     public class OptimalVariantConverter<TLink> :
    ↳ LinksListToSequenceConverterBase<TLink>
8     {
9         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↳ EqualityComparer<TLink>.Default;
10        private static readonly Comparer<TLink> _comparer =
    ↳ Comparer<TLink>.Default;
11
12        private readonly IConverter<IList<TLink>>
    ↳ _sequenceToItsLocalElementLevelsConverter;
13
14        public OptimalVariantConverter(ILinks<TLink> links,
    ↳ IConverter<IList<TLink>> sequenceToItsLocalElementLevelsConverter)
    ↳ : base(links)
15        => _sequenceToItsLocalElementLevelsConverter =
    ↳ sequenceToItsLocalElementLevelsConverter;
16
17        public override TLink Convert(IList<TLink> sequence)
18        {
19            var length = sequence.Count;
20            if (length == 1)

```

```

21        {
22            return sequence[0];
23        }
24        var links = Links;
25        if (length == 2)
26        {
27            return links.GetOrCreate(sequence[0], sequence[1]);
28        }
29        sequence = sequence.ToArray();
30        var levels =
    ↳ _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
31        while (length > 2)
32        {
33            var levelRepeat = 1;
34            var currentLevel = levels[0];
35            var previousLevel = levels[0];
36            var skipOnce = false;
37            var w = 0;
38            for (var i = 1; i < length; i++)
39            {
40                if (_equalityComparer.Equals(currentLevel, levels[i]))
41                {
42                    levelRepeat++;
43                    skipOnce = false;
44                    if (levelRepeat == 2)
45                    {
46                        sequence[w] = links.GetOrCreate(sequence[i - 1],
    ↳ sequence[i]);
47                        var newLevel = i >= length - 1 ?
    ↳ GetPreviousLowerThanCurrentOrCurrent(previousLevel,
    ↳ currentLevel)
    ↳ :
    ↳ i < 2 ?
    ↳ GetNextLowerThanCurrentOrCurrent(currentLevel,
    ↳ levels[i + 1]) :
    ↳ GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
    ↳ currentLevel, levels[i +
    ↳ 1]);
48                        levels[w] = newLevel;
49                        previousLevel = currentLevel;
50                        w++;
51                        levelRepeat = 0;
52                        skipOnce = true;
53                    }
54                    else if (i == length - 1)
55                    {
56                        sequence[w] = sequence[i];
57                        levels[w] = levels[i];
58                        w++;
59                    }
60                }
61                else
62                {
63                    currentLevel = levels[i];
64                    levelRepeat = 1;
65                    if (skipOnce)
66                    {
67                        skipOnce = false;
68                    }
69                    else
70                    {
71                        sequence[w] = sequence[i - 1];
72                        levels[w] = levels[i - 1];
73                        previousLevel = levels[w];
74                        w++;
75                    }
76                }
77            }
78        }
79    }

```

```

80         if (i == length - 1)
81         {
82             sequence[w] = sequence[i];
83             levels[w] = levels[i];
84             w++;
85         }
86     }
87 }
88 length = w;
89 }
90 return links.GetOrCreate(sequence[0], sequence[1]);
91 }
92
93 private static TLink
94     ↪ GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
95     ↪ current, TLink next)
96 {
97     return _comparer.Compare(previous, next) > 0
98         ? _comparer.Compare(previous, current) < 0 ? previous : current
99         : _comparer.Compare(next, current) < 0 ? next : current;
100 }
101
102 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current,
103     ↪ TLink next) => _comparer.Compare(next, current) < 0 ? next :
104     ↪ current;
105
106 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink
107     ↪ previous, TLink current) => _comparer.Compare(previous, current) <
108     ↪ 0 ? previous : current;
109 }
110 }

```

./Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Sequences.Converters
5 {
6     public class SequenceToItsLocalElementLevelsConverter<TLink> :
7     ↪ LinksOperatorBase<TLink>, IConverter<IList<TLink>>
8     {
9         private static readonly Comparer<TLink> _comparer =
10         ↪ Comparer<TLink>.Default;
11         private readonly IConverter<Doublet<TLink>, TLink>
12         ↪ _linkToItsFrequencyToNumberConverter;
13         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
14         ↪ IConverter<Doublet<TLink>, TLink>
15         ↪ linkToItsFrequencyToNumberConverter) : base(links) =>
16         ↪ _linkToItsFrequencyToNumberConverter =
17         ↪ linkToItsFrequencyToNumberConverter;
18         public IList<TLink> Convert(IList<TLink> sequence)
19         {
20             var levels = new TLink[sequence.Count];
21             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
22             for (var i = 1; i < sequence.Count - 1; i++)
23             {
24                 var previous = GetFrequencyNumber(sequence[i - 1],
25                 ↪ sequence[i]);
26                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
27                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous :
28                 ↪ next;
29             }
30             levels[levels.Length - 1] =
31             ↪ GetFrequencyNumber(sequence[sequence.Count - 2],
32             ↪ sequence[sequence.Count - 1]);
33             return levels;
34         }
35     }
36 }

```

```

23     }
24
25     public TLink GetFrequencyNumber(TLink source, TLink target) =>
26     ↪ _linkToItsFrequencyToNumberConverter.Convert(new
27     ↪ Doublet<TLink>(source, target));
28 }
29 }

```

./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs

```

1 using Platform.Interfaces;
2
3 namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
4 {
5     public class DefaultSequenceElementCreteriaMatcher<TLink> :
6     ↪ LinksOperatorBase<TLink>, ICreteriaMatcher<TLink>
7     {
8         public DefaultSequenceElementCreteriaMatcher(ILinks<TLink> links) :
9         ↪ base(links) { }
10         public bool IsMatched(TLink argument) =>
11         ↪ Links.IsPartialPoint(argument);
12     }
13 }

```

./Sequences/CreteriaMatchers/MarkedSequenceCreteriaMatcher.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
5 {
6     public class MarkedSequenceCreteriaMatcher<TLink> : ICreteriaMatcher<TLink>
7     {
8         private static readonly EqualityComparer<TLink> _equalityComparer =
9         ↪ EqualityComparer<TLink>.Default;
10
11         private readonly ILinks<TLink> _links;
12         private readonly TLink _sequenceMarkerLink;
13
14         public MarkedSequenceCreteriaMatcher(ILinks<TLink> links, TLink
15         ↪ sequenceMarkerLink)
16         {
17             _links = links;
18             _sequenceMarkerLink = sequenceMarkerLink;
19
20             public bool IsMatched(TLink sequenceCandidate)
21             => _equalityComparer.Equals(_links.GetSource(sequenceCandidate),
22             ↪ _sequenceMarkerLink)
23             || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarke
24             ↪ rLink, sequenceCandidate),
25             ↪ _links.Constants.Null);
26         }
27     }
28 }

```

./Sequences/DefaultSequenceAppender.cs

```

1 using System.Collections.Generic;
2 using Platform.Collections.Stacks;
3 using Platform.Data.Doublets.Sequences.HeightProviders;
4 using Platform.Data.Sequences;
5
6 namespace Platform.Data.Doublets.Sequences
7 {
8     public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
9     ↪ ISequenceAppender<TLink>
10     {
11     }
12 }

```

```

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

```

./Sequences/DuplicateSegmentsCounter.cs

```

1 using System.Collections.Generic;
2 using System.Linq;
3 using Platform.Interfaces;
4
5 namespace Platform.Data.Doublets.Sequences
6 {
7     public class DuplicateSegmentsCounter<TLink> : ICounter<int>
8     {
9         private readonly IProvider<IList<KeyValuePair<IList<TLink>,
10     ↳ IList<TLink>>>> _duplicateFragmentsProvider;
11         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
12     ↳ nk>, IList<TLink>>>> duplicateFragmentsProvider) =>
13     ↳ _duplicateFragmentsProvider = duplicateFragmentsProvider;
14         public int Count() => _duplicateFragmentsProvider.Get().Sum(x =>
15     ↳ x.Value.Count);
16     }
17 }

```

./Sequences/DuplicateSegmentsProvider.cs

```

1 using System;
2 using System.Linq;
3 using System.Collections.Generic;
4 using Platform.Interfaces;
5 using Platform.Collections;
6 using Platform.Collections.Lists;
7 using Platform.Collections.Segments;
8 using Platform.Collections.Segments.Walkers;
9 using Platform.Helpers;
10 using Platform.Helpers.Singletons;
11 using Platform.Numbers;
12 using Platform.Data.Sequences;
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     public class DuplicateSegmentsProvider<TLink> :
    ↳ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
    ↳ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
17     {
18         private readonly ILinks<TLink> _links;
19         private readonly ISequences<TLink> _sequences;
20         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
21         private BitString _visited;
22
23         private class ItemEquilityComparer :
    ↳ IEquityComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
24         {
25             private readonly IListEqualityComparer<TLink> _listComparer;
26             public ItemEquilityComparer() => _listComparer =
    ↳ Default<IListEqualityComparer<TLink>>.Instance;
27             public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
28     ↳ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
29     ↳ _listComparer.Equals(left.Key, right.Key) &&
30     ↳ _listComparer.Equals(left.Value, right.Value);
31             public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>>
32     ↳ pair) =>
33     ↳ HashHelpers.Generate(_listComparer.GetHashCode(pair.Key),
34     ↳ _listComparer.GetHashCode(pair.Value));
35         }
36
37         private class ItemComparer : IComparer<KeyValuePair<IList<TLink>,
38     ↳ IList<TLink>>>
39         {
40             private readonly IListComparer<TLink> _listComparer;
41
42             public ItemComparer() => _listComparer =
    ↳ Default<IListComparer<TLink>>.Instance;
43
44             public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
45     ↳ KeyValuePair<IList<TLink>, IList<TLink>> right)
46             {
47                 var intermediateResult = _listComparer.Compare(left.Key,
48     ↳ right.Key);
49                 if (intermediateResult == 0)
50                 {
51                     intermediateResult = _listComparer.Compare(left.Value,
52     ↳ right.Value);
53                 }
54                 return intermediateResult;
55             }
56         }
57
58         public DuplicateSegmentsProvider(ILinks<TLink> links,
59     ↳ ISequences<TLink> sequences)
60         : base(minimumStringSegmentLength: 2)
61     {

```

```

51     _links = links;
52     _sequences = sequences;
53 }
54
55 public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
56 {
57     _groups = new HashSet<KeyValuePair<IList<TLink>,
58         ↳ IList<TLink>>>(Default<ItemEqualityComparer>.Instance);
59     var count = _links.Count();
60     _visited = new BitString((long)(Integer<TLink>count + 1);
61     _links.Each(link =>
62     {
63         var linkIndex = _links.GetIndex(link);
64         var linkBitIndex = (long)(Integer<TLink>)linkIndex;
65         if (!_visited.Get(linkBitIndex))
66         {
67             var sequenceElements = new List<TLink>();
68             _sequences.EachPart(sequenceElements.AddAndReturnTrue,
69                 ↳ linkIndex);
70             if (sequenceElements.Count > 2)
71             {
72                 WalkAll(sequenceElements);
73             }
74             return _links.Constants.Continue;
75         });
76         var resultList = _groups.ToList();
77         var comparer = Default<ItemComparer>.Instance;
78         resultList.Sort(comparer);
79
80 #if DEBUG
81         foreach (var item in resultList)
82         {
83             PrintDuplicates(item);
84         }
85 #endif
86         return resultList;
87     }
88
89 protected override Segment<TLink> CreateSegment(IList<TLink> elements,
90     ↳ int offset, int length) => new Segment<TLink>(elements, offset,
91     ↳ length);
92
93 protected override void OnDuplicateFound(Segment<TLink> segment)
94 {
95     var duplicates = CollectDuplicatesForSegment(segment);
96     if (duplicates.Count > 1)
97     {
98         _groups.Add(new KeyValuePair<IList<TLink>,
99             ↳ IList<TLink>>(segment.ToArray(), duplicates));
100     }
101 }
102
103 private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
104 {
105     var duplicates = new List<TLink>();
106     var readAsElement = new HashSet<TLink>();
107     _sequences.Each(sequence =>
108     {
109         duplicates.Add(sequence);
110         readAsElement.Add(sequence);
111         return true; // Continue
112     }, segment);
113     if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
114     {
115         return new List<TLink>();
116     }

```

```

112     foreach (var duplicate in duplicates)
113     {
114         var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
115         _visited.Set(duplicateBitIndex);
116     }
117     if (_sequences is Sequences sequencesExperiments)
118     {
119         var partiallyMatched = sequencesExperiments.GetAllPartiallyMat
120             ↳ chingSequences4((HashSet<ulong>)(object)readAsElement,
121                 ↳ (IList<ulong>)segment);
122         foreach (var partiallyMatchedSequence in partiallyMatched)
123         {
124             TLink sequenceIndex =
125                 ↳ (Integer<TLink>)partiallyMatchedSequence;
126             duplicates.Add(sequenceIndex);
127         }
128     }
129     duplicates.Sort();
130     return duplicates;
131 }
132
133 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>>
134     ↳ duplicatesItem)
135 {
136     if (!(_links is ILinks<ulong> ulongLinks))
137     {
138         return;
139     }
140     var duplicatesKey = duplicatesItem.Key;
141     var keyString =
142         ↳ UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
143     Console.WriteLine($"> {keyString} ({string.Join(", ",
144         ↳ duplicatesKey}))");
145     var duplicatesList = duplicatesItem.Value;
146     for (int i = 0; i < duplicatesList.Count; i++)
147     {
148         ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
149         var formattedSequenceStructure =
150             ↳ ulongLinks.FormatStructure(sequenceIndex, x =>
151                 ↳ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
152                 ↳ UnicodeMap.IsCharLink(link.Index) ?
153                 ↳ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) :
154                 ↳ sb.Append(link.Index));
155         Console.WriteLine(formattedSequenceStructure);
156         var sequenceString =
157             ↳ UnicodeMap.FromSequenceLinkToString(sequenceIndex,
158                 ↳ ulongLinks);
159         Console.WriteLine(sequenceString);
160     }
161     Console.WriteLine();
162 }

```

./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
5 {
6     public class FrequenciesCacheBasedLinkFrequencyIncrementer<TLink> :
7         ↳ IIncrementer<IList<TLink>>
8     {
9         private readonly LinkFrequenciesCache<TLink> _cache;

```

```

9
10     public FrequenciesCacheBasedLinkFrequencyIncrementer(LinkFrequenciesCa
    ↳ che<TLink> cache) => _cache =
    ↳ cache;
11
12     /// <remarks>Sequence itseft is not changed, only frequency of its
    ↳ doublets is incremented.</remarks>
13     public IList<TLink> Increment(IList<TLink> sequence)
14     {
15         _cache.IncrementFrequencies(sequence);
16         return sequence;
17     }
18 }
19 }

```

./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToItsFrequencyNumberConverter.cs

```

1 using Platform.Interfaces;
2
3 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
4 {
5     public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>
    ↳ : IConverter<Doublet<TLink>, TLink>
6     {
7         private readonly LinkFrequenciesCache<TLink> _cache;
8         public FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFreq
    ↳ uenciesCache<TLink> cache) => _cache =
    ↳ cache;
9         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref
    ↳ source).Frequency;
10     }
11 }

```

./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 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
8 {
9     /// <remarks>
10     /// Can be used to operate with many CompressingConverters (to keep global
    ↳ frequencies data between them).
11     /// TODO: Extract interface to implement frequencies storage inside Links
    ↳ storage
12     /// </remarks>
13     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
14     {
15         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↳ EqualityComparer<TLink>.Default;
16         private static readonly Comparer<TLink> _comparer =
    ↳ Comparer<TLink>.Default;
17
18         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>>
    ↳ _doubletsCache;
19         private readonly ICounter<TLink, TLink> _frequencyCounter;
20
21         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink,
    ↳ TLink> frequencyCounter)
22         : base(links)
23         {
24             _doubletsCache = new Dictionary<Doublet<TLink>,
    ↳ LinkFrequency<TLink>>(4096, DoubletComparer<TLink>.Default);
25             _frequencyCounter = frequencyCounter;
26         }

```

```

27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
30     {
31         var doublet = new Doublet<TLink>(source, target);
32         return GetFrequency(ref doublet);
33     }
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
37     {
38         _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
39         return data;
40     }
41
42     public void IncrementFrequencies(IList<TLink> sequence)
43     {
44         for (var i = 1; i < sequence.Count; i++)
45         {
46             IncrementFrequency(sequence[i - 1], sequence[i]);
47         }
48     }
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink
    ↳ target)
52     {
53         var doublet = new Doublet<TLink>(source, target);
54         return IncrementFrequency(ref doublet);
55     }
56
57     public void PrintFrequencies(IList<TLink> sequence)
58     {
59         for (var i = 1; i < sequence.Count; i++)
60         {
61             PrintFrequency(sequence[i - 1], sequence[i]);
62         }
63     }
64
65     public void PrintFrequency(TLink source, TLink target)
66     {
67         var number = GetFrequency(source, target).Frequency;
68         Console.WriteLine("{0},{1} - {2}", source, target, number);
69     }
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink>
    ↳ doublet)
73     {
74         if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink>
    ↳ data))
75         {
76             data.IncrementFrequency();
77         }
78         else
79         {
80             var link = Links.SearchOrDefault(doublet.Source,
    ↳ doublet.Target);
81             data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
82             if (!_equalityComparer.Equals(link, default))
83             {
84                 data.Frequency = ArithmeticHelpers.Add(data.Frequency,
    ↳ _frequencyCounter.Count(link));
85             }
86             _doubletsCache.Add(doublet, data);

```

```

87     }
88     return data;
89 }
90
91 public void ValidateFrequencies()
92 {
93     foreach (var entry in _doubletsCache)
94     {
95         var value = entry.Value;
96         var linkIndex = value.Link;
97         if (!_equalityComparer.Equals(linkIndex, default))
98         {
99             var frequency = value.Frequency;
100             var count = _frequencyCounter.Count(linkIndex);
101             // TODO: Why `frequency` always greater than `count` by 1?
102             if (((_comparer.Compare(frequency, count) > 0) && (_comparer.Compare(ArithmeticHelpers.Subtract(frequency,
103                                     ↪ count), Integer<TLink>.One) > 0)) ||
104                 ((_comparer.Compare(count, frequency) > 0) &&
105                 ↪ (_comparer.Compare(ArithmeticHelpers.Subtract(count,
106                                     ↪ frequency), Integer<TLink>.One) > 0)))
107             {
108                 throw new InvalidOperationException("Frequencies
109                                     ↪ validation failed.");
110             }
111             //else
112             //{
113                 if (value.Frequency > 0)
114                 {
115                     var frequency = value.Frequency;
116                     linkIndex = _createLink(entry.Key.Source,
117                     ↪ entry.Key.Target);
118                     var count = _countLinkFrequency(linkIndex);
119
120                     if ((frequency > count && frequency - count > 1) ||
121                     ↪ (count > frequency && count - frequency > 1))
122                         throw new Exception("Frequencies validation
123                                     ↪ failed.");
124                 }
125             //}
126         }
127     }
128 }
129 }
130 }
131 }
132 }
133 }

```

./Sequences/Frequencies/Cache/LinkFrequency.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Numbers;
3
4 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
5 {
6     public class LinkFrequency<TLink>
7     {
8         public TLink Frequency { get; set; }
9         public TLink Link { get; set; }
10
11         public LinkFrequency(TLink frequency, TLink link)
12         {
13             Frequency = frequency;
14             Link = link;
15         }
16
17         public LinkFrequency() { }
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

20         public void IncrementFrequency() => Frequency =
21             ↪ ArithmeticHelpers<TLink>.Increment(Frequency);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void DecrementFrequency() => Frequency =
25             ↪ ArithmeticHelpers<TLink>.Decrement(Frequency);
26
27     }
28 }

```

./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs

```

1 using Platform.Interfaces;
2
3 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
4 {
5     public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
6         ↪ SequenceSymbolFrequencyOneOffCounter<TLink>
7     {
8         private readonly ICriteriaMatcher<TLink> _markedSequenceMatcher;
9
10         public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
11             ↪ ICriteriaMatcher<TLink> markedSequenceMatcher, TLink sequenceLink,
12             ↪ TLink symbol)
13             : base(links, sequenceLink, symbol)
14             => _markedSequenceMatcher = markedSequenceMatcher;
15
16         public override TLink Count()
17         {
18             if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
19             {
20                 return default;
21             }
22             return base.Count();
23         }
24     }
25 }

```

./Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3 using Platform.Numbers;
4 using Platform.Data.Sequences;
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7 {
8     public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12         private static readonly Comparer<TLink> _comparer =
13             ↪ Comparer<TLink>.Default;
14
15         protected readonly ILinks<TLink> _links;
16         protected readonly TLink _sequenceLink;
17         protected readonly TLink _symbol;
18         protected TLink _total;
19
20         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink
21             ↪ sequenceLink, TLink symbol)
22         {
23             _links = links;
24             _sequenceLink = sequenceLink;
25             _symbol = symbol;
26             _total = default;
27         }
28     }
29 }

```



```

24     }
25
26     public virtual TLink Count()
27     {
28         if (_comparer.Compare(_total, default) > 0)
29         {
30             return _total;
31         }
32         StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource,
33         ↪ _links.GetTarget, IsElement, VisitElement);
34         return _total;
35     }
36
37     private bool IsElement(TLink x) => _equalityComparer.Equals(x,
38     ↪ _symbol) || _links.IsPartialPoint(x); // TODO: Use
39     ↪ SequenceElementCriteriaMatcher instead of IsPartialPoint
40
41     private bool VisitElement(TLink element)
42     {
43         if (_equalityComparer.Equals(element, _symbol))
44         {
45             _total = ArithmeticHelpers.Increment(_total);
46         }
47         return true;
48     }
49 }

```

./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs

```

1  using Platform.Interfaces;
2
3  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
4  {
5      public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> :
6      ↪ ICounter<TLink, TLink>
7      {
8          private readonly ILinks<TLink> _links;
9          private readonly ICriteriaMatcher<TLink> _markedSequenceMatcher;
10
11         public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
12         ↪ ICriteriaMatcher<TLink> markedSequenceMatcher)
13         {
14             _links = links;
15             _markedSequenceMatcher = markedSequenceMatcher;
16         }
17
18         public TLink Count(TLink argument) => new
19         ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
20         ↪ _markedSequenceMatcher, argument).Count();
21     }
22 }

```

./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs

```

1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
5  {
6      public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
7      ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
8      {
9          private readonly ICriteriaMatcher<TLink> _markedSequenceMatcher;
10
11         public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink>
12         ↪ links, ICriteriaMatcher<TLink> markedSequenceMatcher, TLink
13         ↪ symbol) : base(links, symbol)

```

```

11         => _markedSequenceMatcher = markedSequenceMatcher;
12
13         protected override void CountSequenceSymbolFrequency(TLink link)
14         {
15             var symbolFrequencyCounter = new
16             ↪ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
17             ↪ _markedSequenceMatcher, link, _symbol);
18             _total = ArithmeticHelpers.Add(_total,
19             ↪ symbolFrequencyCounter.Count());
20         }
21     }
22 }

```

./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs

```

1  using Platform.Interfaces;
2
3  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
4  {
5      public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink,
6      ↪ TLink>
7      {
8          private readonly ILinks<TLink> _links;
9          public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) =>
10         ↪ _links = links;
11         public TLink Count(TLink symbol) => new
12         ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
13         ↪ symbol).Count();
14     }
15 }

```

./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4
5  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6  {
7      public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> :
8      ↪ ICounter<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11         ↪ EqualityComparer<TLink>.Default;
12         private static readonly Comparer<TLink> _comparer =
13         ↪ Comparer<TLink>.Default;
14
15         protected readonly ILinks<TLink> _links;
16         protected readonly TLink _symbol;
17         protected readonly HashSet<TLink> _visits;
18         protected TLink _total;
19
20         public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
21         ↪ TLink symbol)
22         {
23             _links = links;
24             _symbol = symbol;
25             _visits = new HashSet<TLink>();
26             _total = default;
27         }
28
29         public TLink Count()
30         {
31             if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
32             {
33                 return _total;
34             }

```

```

31     CountCore(_symbol);
32     return _total;
33 }
34
35 private void CountCore(TLink link)
36 {
37     var any = _links.Constants.Any;
38     if (_equalityComparer.Equals(_links.Count(any, link), default))
39     {
40         CountSequenceSymbolFrequency(link);
41     }
42     else
43     {
44         _links.Each(EachElementHandler, any, link);
45     }
46 }
47
48 protected virtual void CountSequenceSymbolFrequency(TLink link)
49 {
50     var symbolFrequencyCounter = new
51         ↪ SequenceSymbolFrequencyOneOffCounter<TLink>(_links, link,
52         ↪ _symbol);
53     _total = ArithmeticHelpers.Add(_total,
54         ↪ symbolFrequencyCounter.Count());
55 }
56
57 private TLink EachElementHandler(IList<TLink> doublet)
58 {
59     var constants = _links.Constants;
60     var doubletIndex = doublet[constants.IndexPart];
61     if (_visits.Add(doubletIndex))
62     {
63         CountCore(doubletIndex);
64     }
65     return constants.Continue;
66 }
67
68 }
69
70 }

```

./Sequences/HeightProviders/CachedSequenceHeightProvider.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Sequences.HeightProviders
5  {
6      public class CachedSequenceHeightProvider<TLink> :
7          ↪ LinksOperatorBase<TLink>, ISequenceHeightProvider<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↪ EqualityComparer<TLink>.Default;
11
12         private readonly TLink _heightPropertyMarker;
13         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
14         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
15         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
16         private readonly IPropertyOperator<TLink, TLink, TLink>
17             ↪ _propertyOperator;
18
19         public CachedSequenceHeightProvider(
20             ILinks<TLink> links,
21             ISequenceHeightProvider<TLink> baseHeightProvider,
22             IConverter<TLink> addressToUnaryNumberConverter,
23             IConverter<TLink> unaryNumberToAddressConverter,
24             TLink heightPropertyMarker,
25             IPropertyOperator<TLink, TLink, TLink> propertyOperator)
26             : base(links)
27         {
28
29         }
30     }
31 }

```

```

25     _heightPropertyMarker = heightPropertyMarker;
26     _baseHeightProvider = baseHeightProvider;
27     _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
28     _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
29     _propertyOperator = propertyOperator;
30 }
31
32 public TLink Get(TLink sequence)
33 {
34     TLink height;
35     var heightValue = _propertyOperator.GetValue(sequence,
36         ↪ _heightPropertyMarker);
37     if (_equalityComparer.Equals(heightValue, default))
38     {
39         height = _baseHeightProvider.Get(sequence);
40         heightValue = _addressToUnaryNumberConverter.Convert(height);
41         _propertyOperator.SetValue(sequence, _heightPropertyMarker,
42             ↪ heightValue);
43     }
44     else
45     {
46         height = _unaryNumberToAddressConverter.Convert(heightValue);
47     }
48     return height;
49 }
50
51 }
52
53 }

```

./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```

1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  namespace Platform.Data.Doublets.Sequences.HeightProviders
5  {
6      public class DefaultSequenceRightHeightProvider<TLink> :
7          ↪ LinksOperatorBase<TLink>, ISequenceHeightProvider<TLink>
8      {
9          private readonly ICreteriaMatcher<TLink> _elementMatcher;
10
11         public DefaultSequenceRightHeightProvider(ILinks<TLink> links,
12             ↪ ICreteriaMatcher<TLink> elementMatcher) : base(links) =>
13             ↪ _elementMatcher = elementMatcher;
14
15         public TLink Get(TLink sequence)
16         {
17             var height = default(TLink);
18             var pairOrElement = sequence;
19             while (!_elementMatcher.IsMatched(pairOrElement))
20             {
21                 pairOrElement = Links.GetTarget(pairOrElement);
22                 height = ArithmeticHelpers.Increment(height);
23             }
24             return height;
25         }
26     }
27 }

```

./Sequences/HeightProviders/ISequenceHeightProvider.cs

```

1  using Platform.Interfaces;
2
3  namespace Platform.Data.Doublets.Sequences.HeightProviders
4  {
5      public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
6      {
7      }
8  }

```

./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.Threading.Synchronization;
8 using Platform.Helpers.Singletons;
9 using LinkIndex = System.UInt64;
10 using Platform.Data.Constants;
11 using Platform.Data.Sequences;
12 using Platform.Data.Doublets.Sequences.Walkers;
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
25     ///   ↳ (подпоследовательностей),
26     ///   ↳ через естественную группировку по unicode типам, все whitespace
27     ///   ↳ вместе, все символы вместе, все числа вместе и т.п.
28     ///   ↳ + использовать ровно сбалансированный вариант, чтобы уменьшать
29     ///   ↳ вложенность (глубину графа)
30     ///
31     /// *х*у - найти все связи между, в последовательностях любой формы, если
32     ///   ↳ не стоит ограничитель на то, что является последовательностью, а что
33     ///   ↳ нет,
34     ///   ↳ то находятся любые структуры связей, которые содержат эти элементы
35     ///   ↳ именно в таком порядке.
36     ///
37     /// Рост последовательности слева и справа.
38     /// Поиск со звёздочкой.
39     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
40     /// так же проблема может быть решена при реализации дистанционных
41     /// триггеров.
42     /// Нужны ли уникальные указатели вообще?
43     /// Что если обращение к информации будет происходить через содержимое
44     ///   ↳ всегда?
45     ///
46     /// Писать тесты.
47     ///
48     /// Можно убрать зависимость от конкретной реализации Links,
49     /// на зависимость от абстрактного элемента, который может быть
50     ///   ↳ представлен несколькими способами.
51     ///
52     /// Можно ли как-то сделать один общий интерфейс
53     ///
54     /// Блокчейн и/или гит для распределённой записи транзакций.
55     /// </remarks>
56     public partial class Sequences : ISequences<ulong> // IList<string>,
57     // ↳ IList<ulong[]> (после завершения реализации Sequences)
58     {
59         private static readonly LinksCombinedConstants<bool, ulong, long>
60         // ↳ _constants = Default<LinksCombinedConstants<bool, ulong,
61         // ↳ long>>.Instance;
```

```
54
55     /// <summary>Возвращает значение ulong, обозначающее любое количество
56     ///   ↳ связей.</summary>
57     public const ulong ZeroOrMany = ulong.MaxValue;
58
59     public SequencesOptions<ulong> Options;
60     public readonly SynchronizedLinks<ulong> Links;
61     public readonly ISynchronization Sync;
62
63     public Sequences(SynchronizedLinks<ulong> links)
64     : this(links, new SequencesOptions<ulong>())
65     {
66     }
67
68     public Sequences(SynchronizedLinks<ulong> links,
69     // ↳ SequencesOptions<ulong> options)
70     {
71         Links = links;
72         Sync = links.SyncRoot;
73         Options = options;
74
75         Options.ValidateOptions();
76         Options.InitOptions(Links);
77     }
78
79     public bool IsSequence(ulong sequence)
80     {
81         return Sync.ExecuteReadOperation(() =>
82         {
83             if (Options.UseSequenceMarker)
84             {
85                 return Options.MarkedSequenceMatcher.IsMatched(sequence);
86             }
87             return !Links.Unsync.IsPartialPoint(sequence);
88         });
89     }
90
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     private ulong GetSequenceByElements(ulong sequence)
93     {
94         if (Options.UseSequenceMarker)
95         {
96             return Links.SearchOrDefault(Options.SequenceMarkerLink,
97             // ↳ sequence);
98         }
99         return sequence;
100     }
101
102     private ulong GetSequenceElements(ulong sequence)
103     {
104         if (Options.UseSequenceMarker)
105         {
106             var linkContents = new UInt64Link(Links.GetLink(sequence));
107             if (linkContents.Source == Options.SequenceMarkerLink)
108             {
109                 return linkContents.Target;
110             }
111             if (linkContents.Target == Options.SequenceMarkerLink)
112             {
113                 return linkContents.Source;
114             }
115             return sequence;
116         }
117     }
118
119     #region Count
120
121     public ulong Count(params ulong[] sequence)
```

```

118 {
119     if (sequence.Length == 0)
120     {
121         return Links.Count(_constants.Any, Options.SequenceMarkerLink,
122             ↪ _constants.Any);
123     }
124     if (sequence.Length == 1) // Первая связь это адрес
125     {
126         if (sequence[0] == _constants.Null)
127         {
128             return 0;
129         }
130         if (sequence[0] == _constants.Any)
131         {
132             return Count();
133         }
134         if (Options.UseSequenceMarker)
135         {
136             return Links.Count(_constants.Any,
137                 ↪ Options.SequenceMarkerLink, sequence[0]);
138         }
139         return Links.Exists(sequence[0]) ? 1UL : 0;
140     }
141     throw new NotImplementedException();
142 }
143 private ulong CountReferences(params ulong[] restrictions)
144 {
145     if (restrictions.Length == 0)
146     {
147         return 0;
148     }
149     if (restrictions.Length == 1) // Первая связь это адрес
150     {
151         if (restrictions[0] == _constants.Null)
152         {
153             return 0;
154         }
155         if (Options.UseSequenceMarker)
156         {
157             var elementsLink = GetSequenceElements(restrictions[0]);
158             var sequenceLink = GetSequenceByElements(elementsLink);
159             if (sequenceLink != _constants.Null)
160             {
161                 return Links.Count(sequenceLink) +
162                     ↪ Links.Count(elementsLink) - 1;
163             }
164             return Links.Count(elementsLink);
165         }
166         return Links.Count(restrictions[0]);
167     }
168     throw new NotImplementedException();
169 }
170 #endregion
171 #region Create
172 public ulong Create(params ulong[] sequence)
173 {
174     return Sync.ExecuteWriteOperation(() =>
175     {
176         if (sequence.IsNullOrEmpty())
177         {
178             return _constants.Null;
179         }
180

```

```

181         Links.EnsureEachLinkExists(sequence);
182         return CreateCore(sequence);
183     });
184 }
185
186 private ulong CreateCore(params ulong[] sequence)
187 {
188     if (Options.UseIndex)
189     {
190         Options.Indexer.Index(sequence);
191     }
192     var sequenceRoot = default(ulong);
193     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
194     {
195         var matches = Each(sequence);
196         if (matches.Count > 0)
197         {
198             sequenceRoot = matches[0];
199         }
200     }
201     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
202     {
203         return CompactCore(sequence);
204     }
205     if (sequenceRoot == default)
206     {
207         sequenceRoot =
208             ↪ Options.LinksToSequenceConverter.Convert(sequence);
209     }
210     if (Options.UseSequenceMarker)
211     {
212         Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink,
213             ↪ sequenceRoot);
214     }
215     return sequenceRoot; // Возвращаем корень последовательности (т.е.
216         ↪ сами элементы)
217 }
218 #endregion
219 #region Each
220 public List<ulong> Each(params ulong[] sequence)
221 {
222     var results = new List<ulong>();
223     Each(results.AddAndReturnTrue, sequence);
224     return results;
225 }
226
227 public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
228 {
229     return Sync.ExecuteReadOperation(() =>
230     {
231         if (sequence.IsNullOrEmpty())
232         {
233             return true;
234         }
235         Links.EnsureEachLinkIsAnyOrExists(sequence);
236         if (sequence.Count == 1)
237         {
238             var link = sequence[0];
239             if (link == _constants.Any)
240             {
241                 return Links.Unsync.Each(_constants.Any,
242                     ↪ _constants.Any, handler);
243             }
244         }
245     });
246 }

```

```

242     }
243     return handler(link);
244 }
245 if (sequence.Count == 2)
246 {
247     return Links.Unsync.Each(sequence[0], sequence[1],
248         ↪ handler);
249 }
250 if (Options.UseIndex && !Options.Indexer.CheckIndex(sequence))
251 {
252     return false;
253 }
254 return EachCore(handler, sequence);
255 });
256 }
257 private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
258 {
259     var matcher = new Matcher(this, sequence, new
260         ↪ HashSet<LinkIndex>(), handler);
261     // TODO: Find out why matcher.HandleFullMatched executed twice for
262     ↪ the same sequence Id.
263     Func<ulong, bool> innerHandler = Options.UseSequenceMarker ?
264         ↪ (Func<ulong, bool>)matcher.HandleFullMatchedSequence :
265         ↪ matcher.HandleFullMatched;
266     //if (sequence.Length >= 2)
267     if (!StepRight(innerHandler, sequence[0], sequence[1]))
268     {
269         return false;
270     }
271     var last = sequence.Count - 2;
272     for (var i = 1; i < last; i++)
273     {
274         if (!PartialStepRight(innerHandler, sequence[i], sequence[i +
275             ↪ 1]))
276         {
277             return false;
278         }
279     }
280     if (sequence.Count >= 3)
281     {
282         if (!StepLeft(innerHandler, sequence[sequence.Count - 2],
283             ↪ sequence[sequence.Count - 1]))
284         {
285             return false;
286         }
287     }
288     return true;
289 }
290 private bool PartialStepRight(Func<ulong, bool> handler, ulong left,
291     ↪ ulong right)
292 {
293     return Links.Unsync.Each(_constants.Any, left, doublet =>
294     {
295         if (!StepRight(handler, doublet, right))
296         {
297             return false;
298         }
299         if (left != doublet)
300         {
301             return PartialStepRight(handler, doublet, right);
302         }
303         return true;
304     });
305 }

```

```

300
301 private bool StepRight(Func<ulong, bool> handler, ulong left, ulong
302     ↪ right) => Links.Unsync.Each(left, _constants.Any, rightStep =>
303     ↪ TryStepRightUp(handler, right, rightStep));
304
305 private bool TryStepRightUp(Func<ulong, bool> handler, ulong right,
306     ↪ ulong stepFrom)
307 {
308     var upStep = stepFrom;
309     var firstSource = Links.Unsync.GetTarget(upStep);
310     while (firstSource != right && firstSource != upStep)
311     {
312         upStep = firstSource;
313         firstSource = Links.Unsync.GetSource(upStep);
314     }
315     if (firstSource == right)
316     {
317         return handler(stepFrom);
318     }
319     return true;
320 }
321
322 private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong
323     ↪ right) => Links.Unsync.Each(_constants.Any, right, leftStep =>
324     ↪ TryStepLeftUp(handler, left, leftStep));
325
326 private bool TryStepLeftUp(Func<ulong, bool> handler, ulong left,
327     ↪ ulong stepFrom)
328 {
329     var upStep = stepFrom;
330     var firstTarget = Links.Unsync.GetSource(upStep);
331     while (firstTarget != left && firstTarget != upStep)
332     {
333         upStep = firstTarget;
334         firstTarget = Links.Unsync.GetTarget(upStep);
335     }
336     if (firstTarget == left)
337     {
338         return handler(stepFrom);
339     }
340     return true;
341 }
342
343 #endregion
344
345 #region Update
346
347 public ulong Update(ulong[] sequence, ulong[] newSequence)
348 {
349     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
350     {
351         return _constants.Null;
352     }
353     if (sequence.IsNullOrEmpty())
354     {
355         return Create(newSequence);
356     }
357     if (newSequence.IsNullOrEmpty())
358     {
359         Delete(sequence);
360         return _constants.Null;
361     }
362     return Sync.ExecuteWriteOperation(() =>
363     {
364         Links.EnsureEachLinkIsAnyOrExists(sequence);
365     });
366 }

```

```

359         Links.EnsureEachLinkExists(newSequence);
360         return UpdateCore(sequence, newSequence);
361     });
362 }
363
364 private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
365 {
366     ulong bestVariant;
367     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
368         ↪ !sequence.EqualTo(newSequence))
369     {
370         bestVariant = CompactCore(newSequence);
371     }
372     else
373     {
374         bestVariant = CreateCore(newSequence);
375     }
376     // TODO: Check all options only ones before loop execution
377     // Возможно нужно две версии Each, возвращающий фактические
378     // ↪ последовательности и с маркером,
379     // или возможно даже возвращать и тот и тот вариант. С другой
380     // ↪ стороны все варианты можно получить имея только фактические
381     // ↪ последовательности.
382     foreach (var variant in Each(sequence))
383     {
384         if (variant != bestVariant)
385         {
386             UpdateOneCore(variant, bestVariant);
387         }
388     }
389     return bestVariant;
390 }
391
392 private void UpdateOneCore(ulong sequence, ulong newSequence)
393 {
394     if (Options.UseGarbageCollection)
395     {
396         var sequenceElements = GetSequenceElements(sequence);
397         var sequenceElementsContents = new
398             ↪ UInt64Link(Links.GetLink(sequenceElements));
399         var sequenceLink = GetSequenceByElements(sequenceElements);
400         var newSequenceElements = GetSequenceElements(newSequence);
401         var newSequenceLink =
402             ↪ GetSequenceByElements(newSequenceElements);
403         if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
404         {
405             if (sequenceLink != _constants.Null)
406             {
407                 Links.Unsync.Merge(sequenceLink, newSequenceLink);
408             }
409             Links.Unsync.Merge(sequenceElements, newSequenceElements);
410         }
411         ClearGarbage(sequenceElementsContents.Source);
412         ClearGarbage(sequenceElementsContents.Target);
413     }
414     else
415     {
416         if (Options.UseSequenceMarker)
417         {
418             var sequenceElements = GetSequenceElements(sequence);
419             var sequenceLink = GetSequenceByElements(sequenceElements);
420             var newSequenceElements = GetSequenceElements(newSequence);
421             var newSequenceLink =
422                 ↪ GetSequenceByElements(newSequenceElements);
423             if (Options.UseCascadeUpdate || CountReferences(sequence)
424                 ↪ == 0)

```

```

417         {
418             if (sequenceLink != _constants.Null)
419             {
420                 Links.Unsync.Merge(sequenceLink, newSequenceLink);
421             }
422             Links.Unsync.Merge(sequenceElements,
423                 ↪ newSequenceElements);
424         }
425     }
426     else
427     {
428         if (Options.UseCascadeUpdate || CountReferences(sequence)
429             ↪ == 0)
430         {
431             Links.Unsync.Merge(sequence, newSequence);
432         }
433     }
434 }
435
436 #endregion
437
438 #region Delete
439
440 public void Delete(params ulong[] sequence)
441 {
442     Sync.ExecuteWriteOperation(() =>
443     {
444         // TODO: Check all options only ones before loop execution
445         foreach (var linkToDelete in Each(sequence))
446         {
447             DeleteOneCore(linkToDelete);
448         }
449     });
450 }
451
452 private void DeleteOneCore(ulong link)
453 {
454     if (Options.UseGarbageCollection)
455     {
456         var sequenceElements = GetSequenceElements(link);
457         var sequenceElementsContents = new
458             ↪ UInt64Link(Links.GetLink(sequenceElements));
459         var sequenceLink = GetSequenceByElements(sequenceElements);
460         if (Options.UseCascadeDelete || CountReferences(link) == 0)
461         {
462             if (sequenceLink != _constants.Null)
463             {
464                 Links.Unsync.Delete(sequenceLink);
465             }
466             Links.Unsync.Delete(link);
467         }
468         ClearGarbage(sequenceElementsContents.Source);
469         ClearGarbage(sequenceElementsContents.Target);
470     }
471     else
472     {
473         if (Options.UseSequenceMarker)
474         {
475             var sequenceElements = GetSequenceElements(link);
476             var sequenceLink = GetSequenceByElements(sequenceElements);
477             if (Options.UseCascadeDelete || CountReferences(link) == 0)
478             {
479                 if (sequenceLink != _constants.Null)
480                 {

```

```

479         Links.Unsync.Delete(sequenceLink);
480     }
481     Links.Unsync.Delete(link);
482 }
483 }
484 else
485 {
486     if (Options.UseCascadeDelete || CountReferences(link) == 0)
487     {
488         Links.Unsync.Delete(link);
489     }
490 }
491 }
492 }
493
494 #endregion
495
496 #region Compactification
497
498 /// <remarks>
499 /// bestVariant можно выбирать по максимальному числу использований,
500 /// но балансированный позволяет гарантировать уникальность (если есть
501   ↳ возможность,
502   ↳ гарантировать его использование в других местах).
503 ///
504 /// Получается этот метод должен игнорировать
505   ↳ Options.EnforceSingleSequenceVersionOnWrite
506 /// </remarks>
507 public ulong Compact(params ulong[] sequence)
508 {
509     return Sync.ExecuteWriteOperation(() =>
510     {
511         if (sequence.IsNullOrEmpty())
512         {
513             return _constants.Null;
514         }
515         Links.EnsureEachLinkExists(sequence);
516         return CompactCore(sequence);
517     });
518 }
519
520 [MethodImpl(MethodImplOptions.AggressiveInlining)]
521 private ulong CompactCore(params ulong[] sequence) =>
522   ↳ UpdateCore(sequence, sequence);
523
524 #endregion
525
526 #region Garbage Collection
527
528 /// <remarks>
529 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted
530   ↳ которое можно определить извне или в унаследованном классе
531 /// </remarks>
532 [MethodImpl(MethodImplOptions.AggressiveInlining)]
533 private bool IsGarbage(ulong link) => link !=
534   ↳ Options.SequenceMarkerLink && !Links.Unsync.IsPartialPoint(link)
535   ↳ && Links.Count(link) == 0;
536
537 private void ClearGarbage(ulong link)
538 {
539     if (IsGarbage(link))
540     {
541         var contents = new UInt64Link(Links.GetLink(link));
542         Links.Unsync.Delete(link);
543         ClearGarbage(contents.Source);
544         ClearGarbage(contents.Target);

```

```

539     }
540 }
541
542 #endregion
543
544 #region Walkers
545
546 public bool EachPart(Func<ulong, bool> handler, ulong sequence)
547 {
548     return Sync.ExecuteReadOperation(() =>
549     {
550         var links = Links.Unsync;
551         var walker = new RightSequenceWalker<ulong>(links);
552         foreach (var part in walker.Walk(sequence))
553         {
554             if (!handler(links.GetIndex(part)))
555             {
556                 return false;
557             }
558         }
559         return true;
560     });
561 }
562
563 public class Matcher : RightSequenceWalker<ulong>
564 {
565     private readonly Sequences _sequences;
566     private readonly IList<LinkIndex> _patternSequence;
567     private readonly HashSet<LinkIndex> _linksInSequence;
568     private readonly HashSet<LinkIndex> _results;
569     private readonly Func<ulong, bool> _stopableHandler;
570     private readonly HashSet<ulong> _readAsElements;
571     private int _filterPosition;
572
573     public Matcher(Sequences sequences, IList<LinkIndex>
574   ↳ patternSequence, HashSet<LinkIndex> results, Func<LinkIndex,
575   ↳ bool> stopableHandler, HashSet<LinkIndex> readAsElements =
576   ↳ null)
577       : base(sequences.Links.Unsync)
578     {
579         _sequences = sequences;
580         _patternSequence = patternSequence;
581         _linksInSequence = new
582   ↳ HashSet<LinkIndex>(patternSequence.Where(x => x !=
583   ↳ _constants.Any && x != ZeroOrMany));
584         _results = results;
585         _stopableHandler = stopableHandler;
586         _readAsElements = readAsElements;
587     }
588
589     protected override bool IsElement(IList<ulong> link) =>
590   ↳ base.IsElement(link) || (_readAsElements != null &&
591   ↳ _readAsElements.Contains(Links.GetIndex(link))) ||
592   ↳ _linksInSequence.Contains(Links.GetIndex(link));
593
594     public bool FullMatch(LinkIndex sequenceToMatch)
595     {
596         _filterPosition = 0;
597         foreach (var part in Walk(sequenceToMatch))
598         {
599             if (!FullMatchCore(Links.GetIndex(part)))
600             {
601                 break;
602             }
603         }
604         return _filterPosition == _patternSequence.Count;
605     }

```

```

599 private bool FullMatchCore(LinkIndex element)
600 {
601     if (_filterPosition == _patternSequence.Count)
602     {
603         _filterPosition = -2; // Длиннее чем нужно
604         return false;
605     }
606     if (_patternSequence[_filterPosition] != _constants.Any
607     && element != _patternSequence[_filterPosition])
608     {
609         _filterPosition = -1;
610         return false; // Начинается/Продолжается иначе
611     }
612     _filterPosition++;
613     return true;
614 }
615
616 public void AddFullMatchedToResults(ulong sequenceToMatch)
617 {
618     if (FullMatch(sequenceToMatch))
619     {
620         _results.Add(sequenceToMatch);
621     }
622 }
623
624 public bool HandleFullMatched(ulong sequenceToMatch)
625 {
626     if (FullMatch(sequenceToMatch) &&
627         → _results.Add(sequenceToMatch))
628     {
629         return _stopableHandler(sequenceToMatch);
630     }
631     return true;
632 }
633
634 public bool HandleFullMatchedSequence(ulong sequenceToMatch)
635 {
636     var sequence =
637         → _sequences.GetSequenceByElements(sequenceToMatch);
638     if (sequence != _constants.Null && FullMatch(sequenceToMatch)
639         → && _results.Add(sequenceToMatch))
640     {
641         return _stopableHandler(sequence);
642     }
643     return true;
644 }
645
646 /// <remarks>
647 /// TODO: Add support for LinksConstants.Any
648 /// </remarks>
649 public bool PartialMatch(LinkIndex sequenceToMatch)
650 {
651     _filterPosition = -1;
652     foreach (var part in Walk(sequenceToMatch))
653     {
654         if (!PartialMatchCore(Links.GetIndex(part)))
655         {
656             break;
657         }
658     }
659     return _filterPosition == _patternSequence.Count - 1;
660 }
661
662 private bool PartialMatchCore(LinkIndex element)
663 {
664     if (_filterPosition == (_patternSequence.Count - 1))

```

```

665 {
666     return false; // Нашлось
667 }
668
669 if (_filterPosition >= 0)
670 {
671     if (element == _patternSequence[_filterPosition + 1])
672     {
673         _filterPosition++;
674     }
675     else
676     {
677         _filterPosition = -1;
678     }
679 }
680
681 if (_filterPosition < 0)
682 {
683     if (element == _patternSequence[0])
684     {
685         _filterPosition = 0;
686     }
687 }
688
689 return true; // Ищем дальше
690 }
691
692 public void AddPartialMatchedToResults(ulong sequenceToMatch)
693 {
694     if (PartialMatch(sequenceToMatch))
695     {
696         _results.Add(sequenceToMatch);
697     }
698 }
699
700 public bool HandlePartialMatched(ulong sequenceToMatch)
701 {
702     if (PartialMatch(sequenceToMatch))
703     {
704         return _stopableHandler(sequenceToMatch);
705     }
706     return true;
707 }
708
709 public void AddAllPartialMatchedToResults(IEnumerable<ulong>
710 → sequencesToMatch)
711 {
712     foreach (var sequenceToMatch in sequencesToMatch)
713     {
714         if (PartialMatch(sequenceToMatch))
715         {
716             _results.Add(sequenceToMatch);
717         }
718     }
719 }
720
721 public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumer
722 → able<ulong>
723 → sequencesToMatch)
724 {
725     foreach (var sequenceToMatch in sequencesToMatch)
726     {
727         if (PartialMatch(sequenceToMatch))
728         {
729             _readAsElements.Add(sequenceToMatch);
730             _results.Add(sequenceToMatch);
731         }
732     }
733 }

```



```

724     }
725 }
726
727 #endregion
728 }
729 }

./Sequences/Sequences.Experiments.cs
1  using System;
2  using LinkIndex = System.UInt64;
3  using System.Collections.Generic;
4  using Stack = System.Collections.Generic.Stack<ulong>;
5  using System.Linq;
6  using System.Text;
7  using Platform.Collections;
8  using Platform.Numbers;
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
14 namespace Platform.Data.Doublets.Sequences
15 {
16     partial class Sequences
17     {
18         #region Create All Variants (Not Practical)
19
20         /// <remarks>
21         /// Number of links that is needed to generate all variants for
22         /// sequence of length N corresponds to https://oeis.org/A014143/list
23         /// ↪ sequence.
24         /// </remarks>
25         public ulong[] CreateAllVariants2(ulong[] sequence)
26         {
27             return Sync.ExecuteWriteOperation(() =>
28             {
29                 if (sequence.IsNullOrEmpty())
30                 {
31                     return new ulong[0];
32                 }
33                 Links.EnsureEachLinkExists(sequence);
34                 if (sequence.Length == 1)
35                 {
36                     return sequence;
37                 }
38                 return CreateAllVariants2Core(sequence, 0, sequence.Length -
39                     ↪ 1);
40             });
41         }
42
43         private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt,
44             ↪ long stopAt)
45         {
46             #if DEBUG
47                 if ((stopAt - startAt) < 0)
48                 {
49                     throw new ArgumentOutOfRangeException(nameof(startAt),
50                         ↪ "startAt должен быть меньше или равен stopAt");
51                 }
52             #endif
53             if ((stopAt - startAt) == 0)
54             {
55                 return new[] { sequence[startAt] };
56             }
57             if ((stopAt - startAt) == 1)
58             {
59             }
60         }

```

```

55         return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt],
60             ↪ sequence[stopAt]) };
61     }
62     var variants = new ulong[(ulong)MathHelpers.Catalan(stopAt -
63         ↪ startAt)];
64     var last = 0;
65     for (var splitter = startAt; splitter < stopAt; splitter++)
66     {
67         var left = CreateAllVariants2Core(sequence, startAt, splitter);
68         var right = CreateAllVariants2Core(sequence, splitter + 1,
69             ↪ 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.CreateAndUpdate(left[i],
75                     ↪ right[j]);
76                 if (variant == _constants.Null)
77                 {
78                     throw new NotImplementedException("Creation
79                         ↪ cancellation is not implemented.");
80                 }
81                 variants[last++] = variant;
82             }
83         }
84     }
85     return variants;
86 }
87
88 public List<ulong> CreateAllVariants1(params ulong[] sequence)
89 {
90     return Sync.ExecuteWriteOperation(() =>
91     {
92         if (sequence.IsNullOrEmpty())
93         {
94             return new List<ulong>();
95         }
96         Links.Unsync.EnsureEachLinkExists(sequence);
97         if (sequence.Length == 1)
98         {
99             return new List<ulong> { sequence[0] };
100         }
101         var results = new
102             ↪ List<ulong>((int)MathHelpers.Catalan(sequence.Length));
103         return CreateAllVariants1Core(sequence, results);
104     });
105 }
106
107 private List<ulong> CreateAllVariants1Core(ulong[] sequence,
108     ↪ List<ulong> results)
109 {
110     if (sequence.Length == 2)
111     {
112         var link = Links.Unsync.CreateAndUpdate(sequence[0],
113             ↪ sequence[1]);
114         if (link == _constants.Null)
115         {
116             throw new NotImplementedException("Creation cancellation
117                 ↪ is not implemented.");
118         }
119         results.Add(link);
120         return results;
121     }
122     var innerSequenceLength = sequence.Length - 1;
123     var innerSequence = new ulong[innerSequenceLength];

```

```

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

```

```

175                     }
176                     for (var isi = linkIndex + 1; isi <
177                     ↪ innerSequenceLength; isi++)
178                     {
179                         innerSequence[isi] = sequence[isi + 1];
180                     }
181                     innerSequence[linkIndex] = doublet;
182                     Each1(handler, innerSequence);
183                     return _constants.Continue;
184                 });
185             }
186         }
187     }
188 }
189 public HashSet<ulong> EachPart(params ulong[] sequence)
190 {
191     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
192     EachPartCore(link =>
193     {
194         if (!visitedLinks.Contains(link))
195         {
196             visitedLinks.Add(link); // изучить почему случаются повторы
197         }
198         return true;
199     }, sequence);
200     return visitedLinks;
201 }
202 public void EachPart(Func<ulong, bool> handler, params ulong[]
203     ↪ sequence)
204 {
205     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
206     EachPartCore(link =>
207     {
208         if (!visitedLinks.Contains(link))
209         {
210             visitedLinks.Add(link); // изучить почему случаются повторы
211             return handler(link);
212         }
213         return true;
214     }, sequence);
215 }
216 private void EachPartCore(Func<ulong, bool> handler, params ulong[]
217     ↪ sequence)
218 {
219     if (sequence.IsNullOrEmpty())
220     {
221         return;
222     }
223     Links.EnsureEachLinkIsAnyOrExists(sequence);
224     if (sequence.Length == 1)
225     {
226         var link = sequence[0];
227         if (link > 0)
228         {
229             handler(link);
230         }
231         else
232         {
233             Links.Each(_constants.Any, _constants.Any, handler);
234         }
235     }
236     else if (sequence.Length == 2)

```

```

237 {
238     // _links.Each(sequence[0], sequence[1], handler);
239     //   o_|      x_o ...
240     // x_|      |___|
241     Links.Each(sequence[1], _constants.Any, doublet =>
242     {
243         var match = Links.SearchOrDefault(sequence[0], doublet);
244         if (match != _constants.Null)
245         {
246             handler(match);
247         }
248         return true;
249     });
250     // |_x      ... x_o
251     // |_o      |___|
252     Links.Each(_constants.Any, sequence[0], doublet =>
253     {
254         var match = Links.SearchOrDefault(doublet, sequence[1]);
255         if (match != 0)
256         {
257             handler(match);
258         }
259         return true;
260     });
261     //      .x o..
262     //      |___|
263     PartialStepRight(x => handler(x), sequence[0], sequence[1]);
264 }
265 else
266 {
267     // TODO: Implement other variants
268     return;
269 }
270 }
271
272 private void PartialStepRight(Action<ulong> handler, ulong left, ulong
↵ right)
273 {
274     Links.Unsync.Each(_constants.Any, left, doublet =>
275     {
276         StepRight(handler, doublet, right);
277         if (left != doublet)
278         {
279             PartialStepRight(handler, doublet, right);
280         }
281         return true;
282     });
283 }
284
285 private void StepRight(Action<ulong> handler, ulong left, ulong right)
286 {
287     Links.Unsync.Each(left, _constants.Any, rightStep =>
288     {
289         TryStepRightUp(handler, right, rightStep);
290         return true;
291     });
292 }
293
294 private void TryStepRightUp(Action<ulong> handler, ulong right, ulong
↵ stepFrom)
295 {
296     var upStep = stepFrom;
297     var firstSource = Links.Unsync.GetTarget(upStep);
298     while (firstSource != right && firstSource != upStep)
299     {
300         upStep = firstSource;

```

```

301         firstSource = Links.Unsync.GetSource(upStep);
302     }
303     if (firstSource == right)
304     {
305         handler(stepFrom);
306     }
307 }
308
309 // TODO: Test
310 private void PartialStepLeft(Action<ulong> handler, ulong left, ulong
↵ right)
311 {
312     Links.Unsync.Each(right, _constants.Any, doublet =>
313     {
314         StepLeft(handler, left, doublet);
315         if (right != doublet)
316         {
317             PartialStepLeft(handler, left, doublet);
318         }
319         return true;
320     });
321 }
322
323 private void StepLeft(Action<ulong> handler, ulong left, ulong right)
324 {
325     Links.Unsync.Each(_constants.Any, right, leftStep =>
326     {
327         TryStepLeftUp(handler, left, leftStep);
328         return true;
329     });
330 }
331
332 private void TryStepLeftUp(Action<ulong> handler, ulong left, ulong
↵ stepFrom)
333 {
334     var upStep = stepFrom;
335     var firstTarget = Links.Unsync.GetSource(upStep);
336     while (firstTarget != left && firstTarget != upStep)
337     {
338         upStep = firstTarget;
339         firstTarget = Links.Unsync.GetTarget(upStep);
340     }
341     if (firstTarget == left)
342     {
343         handler(stepFrom);
344     }
345 }
346
347 private bool StartsWith(ulong sequence, ulong link)
348 {
349     var upStep = sequence;
350     var firstSource = Links.Unsync.GetSource(upStep);
351     while (firstSource != link && firstSource != upStep)
352     {
353         upStep = firstSource;
354         firstSource = Links.Unsync.GetSource(upStep);
355     }
356     return firstSource == link;
357 }
358
359 private bool EndsWith(ulong sequence, ulong link)
360 {
361     var upStep = sequence;
362     var lastTarget = Links.Unsync.GetTarget(upStep);
363     while (lastTarget != link && lastTarget != upStep)
364     {

```

```

365         upStep = lastTarget;
366         lastTarget = Links.Unsync.GetTarget(upStep);
367     }
368     return lastTarget == link;
369 }
370
371 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
372 {
373     return Sync.ExecuteReadOperation(() =>
374     {
375         var results = new List<ulong>();
376         if (sequence.Length > 0)
377         {
378             Links.EnsureEachLinkExists(sequence);
379             var firstElement = sequence[0];
380             if (sequence.Length == 1)
381             {
382                 results.Add(firstElement);
383                 return results;
384             }
385             if (sequence.Length == 2)
386             {
387                 var doublet = Links.SearchOrDefault(firstElement,
388                 ↪ sequence[1]);
389                 if (doublet != _constants.Null)
390                 {
391                     results.Add(doublet);
392                 }
393                 return results;
394             }
395             var linksInSequence = new HashSet<ulong>(sequence);
396             void handler(ulong result)
397             {
398                 var filterPosition = 0;
399                 StopableSequenceWalker.WalkRight(result,
400                 ↪ Links.Unsync.GetSource, Links.Unsync.GetTarget,
401                 ↪ x => linksInSequence.Contains(x) ||
402                 ↪ Links.Unsync.GetTarget(x) == x, x =>
403                 {
404                     if (filterPosition == sequence.Length)
405                     {
406                         filterPosition = -2; // Длиннее чем нужно
407                         return false;
408                     }
409                     if (x != sequence[filterPosition])
410                     {
411                         filterPosition = -1;
412                         return false; // Начинается иначе
413                     }
414                     filterPosition++;
415                     return true;
416                 });
417             if (filterPosition == sequence.Length)
418             {
419                 results.Add(result);
420             }
421         }
422         if (sequence.Length >= 2)
423         {
424             StepRight(handler, sequence[0], sequence[1]);
425         }
426         var last = sequence.Length - 2;
427         for (var i = 1; i < last; i++)
428         {

```

```

427             PartialStepRight(handler, sequence[i], sequence[i +
428             ↪ 1]);
429         }
430         if (sequence.Length >= 3)
431         {
432             StepLeft(handler, sequence[sequence.Length - 2],
433             ↪ sequence[sequence.Length - 1]);
434         }
435     }
436     return results;
437 });
438 }
439
440 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
441 {
442     return Sync.ExecuteReadOperation(() =>
443     {
444         var results = new HashSet<ulong>();
445         if (sequence.Length > 0)
446         {
447             Links.EnsureEachLinkExists(sequence);
448             var firstElement = sequence[0];
449             if (sequence.Length == 1)
450             {
451                 results.Add(firstElement);
452                 return results;
453             }
454             if (sequence.Length == 2)
455             {
456                 var doublet = Links.SearchOrDefault(firstElement,
457                 ↪ sequence[1]);
458                 if (doublet != _constants.Null)
459                 {
460                     results.Add(doublet);
461                 }
462                 return results;
463             }
464             var matcher = new Matcher(this, sequence, results, null);
465             if (sequence.Length >= 2)
466             {
467                 StepRight(matcher.AddFullMatchedToResults,
468                 ↪ sequence[0], sequence[1]);
469             }
470             var last = sequence.Length - 2;
471             for (var i = 1; i < last; i++)
472             {
473                 PartialStepRight(matcher.AddFullMatchedToResults,
474                 ↪ sequence[i], sequence[i + 1]);
475             }
476             if (sequence.Length >= 3)
477             {
478                 StepLeft(matcher.AddFullMatchedToResults,
479                 ↪ sequence[sequence.Length - 2],
480                 ↪ sequence[sequence.Length - 1]);
481             }
482         }
483     }
484     return results;
485 });
486 }
487
488 public const int MaxSequenceFormatSize = 200;
489
490 public string FormatSequence(LinkIndex sequenceLink, params
491 ↪ LinkIndex[] knownElements) => FormatSequence(sequenceLink, (sb, x)
492 ↪ => sb.Append(x), true, knownElements);

```

```

483 public string FormatSequence(LinkIndex sequenceLink,
484     Action<StringBuilder, LinkIndex> elementToString, bool
    ↪ insertComma, params LinkIndex[] knownElements) =>
    ↪ Links.SyncRoot.ExecuteReadOperation(() =>
    ↪ FormatSequence(Links.Unsync, sequenceLink, elementToString,
    ↪ insertComma, knownElements));

485 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex
486     ↪ sequenceLink, Action<StringBuilder, LinkIndex> elementToString,
    ↪ bool insertComma, params LinkIndex[] knownElements)
{
    ↪ var linksInSequence = new HashSet<ulong>(knownElements);
    ↪ //var entered = new HashSet<ulong>();
    ↪ var sb = new StringBuilder();
    ↪ sb.Append('{');
    ↪ if (links.Exists(sequenceLink))
    ↪ {
        ↪ StopableSequenceWalker.WalkRight(sequenceLink,
        ↪     ↪ links.GetSource, links.GetTarget,
        ↪     ↪ x => linksInSequence.Contains(x) ||
        ↪     ↪ links.IsPartialPoint(x), element => //
        ↪     ↪ entered.AddAndReturnVoid, x => { },
        ↪     ↪ entered.DoNotContains
        ↪     {
        ↪         ↪ if (insertComma && sb.Length > 1)
        ↪         ↪ {
        ↪             ↪ sb.Append(',');
        ↪         ↪ }
        ↪         ↪ //if (entered.Contains(element))
        ↪         ↪ //{
        ↪         ↪ //     sb.Append('{');
        ↪         ↪ //     elementToString(sb, element);
        ↪         ↪ //     sb.Append('');
        ↪         ↪ //}
        ↪         ↪ //else
        ↪         ↪ elementToString(sb, element);
        ↪         ↪ if (sb.Length < MaxSequenceFormatSize)
        ↪         ↪ {
        ↪             ↪ return true;
        ↪         ↪ }
        ↪         ↪ sb.Append(insertComma ? ", ..." : "...");
        ↪         ↪ return false;
        ↪     }
    ↪ }
    ↪ sb.Append('}');
    ↪ return sb.ToString();
}

519 public string SafeFormatSequence(LinkIndex sequenceLink, params
520     ↪ LinkIndex[] knownElements) => SafeFormatSequence(sequenceLink,
    ↪ (sb, x) => sb.Append(x), true, knownElements);

521 public string SafeFormatSequence(LinkIndex sequenceLink,
    ↪ Action<StringBuilder, LinkIndex> elementToString, bool
    ↪ insertComma, params LinkIndex[] knownElements) =>
    ↪ Links.SyncRoot.ExecuteReadOperation(() =>
    ↪ SafeFormatSequence(Links.Unsync, sequenceLink, elementToString,
    ↪ insertComma, knownElements));

522 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex
523     ↪ sequenceLink, Action<StringBuilder, LinkIndex> elementToString,
    ↪ bool insertComma, params LinkIndex[] knownElements)
{
    ↪ var linksInSequence = new HashSet<ulong>(knownElements);

```

```

528 var entered = new HashSet<ulong>();
529 var sb = new StringBuilder();
530 sb.Append('{');
531 if (links.Exists(sequenceLink))
532 {
533     StopableSequenceWalker.WalkRight(sequenceLink,
    ↪     ↪ links.GetSource, links.GetTarget,
    ↪     ↪ x => linksInSequence.Contains(x) || links.IsFullPoint(x),
    ↪     ↪ entered.AddAndReturnVoid, x => { },
    ↪     ↪ entered.DoNotContains, element =>
    ↪     {
        ↪         ↪ if (insertComma && sb.Length > 1)
        ↪         ↪ {
        ↪             ↪ sb.Append(',');
        ↪         ↪ }
        ↪         ↪ if (entered.Contains(element))
        ↪         ↪ {
        ↪             ↪ sb.Append('{');
        ↪             ↪ elementToString(sb, element);
        ↪             ↪ sb.Append('');
        ↪         ↪ }
        ↪         ↪ else
        ↪         ↪ {
        ↪             ↪ elementToString(sb, element);
        ↪         ↪ }
        ↪         ↪ if (sb.Length < MaxSequenceFormatSize)
        ↪         ↪ {
        ↪             ↪ return true;
        ↪         ↪ }
        ↪         ↪ sb.Append(insertComma ? ", ..." : "...");
        ↪         ↪ return false;
        ↪     }
    }
    ↪ sb.Append('}');
    ↪ return sb.ToString();
}

559 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[]
560     ↪ sequence)
{
    ↪ return Sync.ExecuteReadOperation(() =>
    ↪ {
        ↪ if (sequence.Length > 0)
        ↪ {
        ↪     ↪ Links.EnsureEachLinkExists(sequence);
        ↪     ↪ var results = new HashSet<ulong>();
        ↪     ↪ for (var i = 0; i < sequence.Length; i++)
        ↪     ↪ {
        ↪         ↪ AllUsagesCore(sequence[i], results);
        ↪     ↪ }
        ↪     ↪ var filteredResults = new List<ulong>();
        ↪     ↪ var linksInSequence = new HashSet<ulong>(sequence);
        ↪     ↪ foreach (var result in results)
        ↪     ↪ {
        ↪         ↪ var filterPosition = -1;
        ↪         ↪ StopableSequenceWalker.WalkRight(result,
        ↪         ↪     ↪ Links.Unsync.GetSource, Links.Unsync.GetTarget,
        ↪         ↪     ↪ x => linksInSequence.Contains(x) ||
        ↪         ↪     ↪ Links.Unsync.GetTarget(x) == x, x =>
        ↪         ↪     {
        ↪             ↪ if (filterPosition == (sequence.Length - 1))
        ↪             ↪ {
        ↪                 ↪ return false;
        ↪             ↪ }
        ↪             ↪ if (filterPosition >= 0)

```

```

587         {
588             if (x == sequence[filterPosition + 1])
589             {
590                 filterPosition++;
591             }
592             else
593             {
594                 return false;
595             }
596         }
597         if (filterPosition < 0)
598         {
599             if (x == sequence[0])
600             {
601                 filterPosition = 0;
602             }
603         }
604         return true;
605     });
606     if (filterPosition == (sequence.Length - 1))
607     {
608         filteredResults.Add(result);
609     }
610     return filteredResults;
611 }
612 return new List<ulong>();
613 });
614 }
615 }
616 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[]
617 ↪ sequence)
618 {
619     return Sync.ExecuteReadOperation(() =>
620     {
621         if (sequence.Length > 0)
622         {
623             Links.EnsureEachLinkExists(sequence);
624             var results = new HashSet<ulong>();
625             for (var i = 0; i < sequence.Length; i++)
626             {
627                 AllUsagesCore(sequence[i], results);
628             }
629             var filteredResults = new HashSet<ulong>();
630             var matcher = new Matcher(this, sequence, filteredResults,
631 ↪ null);
632             matcher.AddAllPartialMatchedToResults(results);
633             return filteredResults;
634         }
635         return new HashSet<ulong>();
636     });
637 }
638 public bool GetAllPartiallyMatchingSequences2(Func<ulong, bool>
639 ↪ handler, params ulong[] sequence)
640 {
641     return Sync.ExecuteReadOperation(() =>
642     {
643         if (sequence.Length > 0)
644         {
645             Links.EnsureEachLinkExists(sequence);
646             var results = new HashSet<ulong>();
647             var filteredResults = new HashSet<ulong>();
648             var matcher = new Matcher(this, sequence, filteredResults,
649 ↪ handler);

```

```

649         for (var i = 0; i < sequence.Length; i++)
650         {
651             if (!AllUsagesCore1(sequence[i], results,
652 ↪ matcher.HandlePartialMatched))
653             {
654                 return false;
655             }
656             return true;
657         }
658         return true;
659     });
660 }
661 }
662 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params
663 ↪ ulong[] sequence)
664 //{
665 //    return Sync.ExecuteReadOperation(() =>
666 //    {
667 //        if (sequence.Length > 0)
668 //        {
669 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
670 //
671 //            var firstResults = new HashSet<ulong>();
672 //            var lastResults = new HashSet<ulong>();
673 //
674 //            var first = sequence.First(x => x != LinksConstants.Any);
675 //            var last = sequence.Last(x => x != LinksConstants.Any);
676 //
677 //            AllUsagesCore(first, firstResults);
678 //            AllUsagesCore(last, lastResults);
679 //
680 //            firstResults.IntersectWith(lastResults);
681 //
682 //            //for (var i = 0; i < sequence.Length; i++)
683 //            //    AllUsagesCore(sequence[i], results);
684 //
685 //            var filteredResults = new HashSet<ulong>();
686 //            var matcher = new Matcher(this, sequence,
687 ↪ filteredResults, null);
688 //            matcher.AddAllPartialMatchedToResults(firstResults);
689 //            return filteredResults;
690 //        }
691 //        return new HashSet<ulong>();
692 //    });
693 //}
694 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[]
695 ↪ sequence)
696 {
697     return Sync.ExecuteReadOperation(() =>
698     {
699         if (sequence.Length > 0)
700         {
701             Links.EnsureEachLinkIsAnyOrExists(sequence);
702             var firstResults = new HashSet<ulong>();
703             var lastResults = new HashSet<ulong>();
704             var first = sequence.First(x => x != _constants.Any);
705             var last = sequence.Last(x => x != _constants.Any);
706             AllUsagesCore(first, firstResults);
707             AllUsagesCore(last, lastResults);
708             firstResults.IntersectWith(lastResults);
709             //for (var i = 0; i < sequence.Length; i++)
710             //    AllUsagesCore(sequence[i], results);

```

```

710         var filteredResults = new HashSet<ulong>();
711         var matcher = new Matcher(this, sequence, filteredResults,
        ↪ null);
712         matcher.AddAllPartialMatchedToResults(firstResults);
713         return filteredResults;
714     }
715     return new HashSet<ulong>();
716 });
717 }
718
719 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong>
        ↪ readAsElements, IList<ulong> sequence)
720 {
721     return Sync.ExecuteReadOperation(() =>
722     {
723         if (sequence.Count > 0)
724         {
725             Links.EnsureEachLinkExists(sequence);
726             var results = new HashSet<LinkIndex>();
727             //var nextResults = new HashSet<ulong>();
728             //for (var i = 0; i < sequence.Length; i++)
729             //{
730                 AllUsagesCore(sequence[i], nextResults);
731                 // if (results.IsNullOrEmpty())
732                 // {
733                     results = nextResults;
734                     nextResults = new HashSet<ulong>();
735                 }
736                 // else
737                 // {
738                     results.IntersectWith(nextResults);
739                     nextResults.Clear();
740                 }
741             //}
742             var collector1 = new AllUsagesCollector1(Links.Unsync,
        ↪ results);
743             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
744             var next = new HashSet<ulong>();
745             for (var i = 1; i < sequence.Count; i++)
746             {
747                 var collector = new AllUsagesCollector1(Links.Unsync,
        ↪ next);
748                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
749
750                 results.IntersectWith(next);
751                 next.Clear();
752             }
753             var filteredResults = new HashSet<ulong>();
754             var matcher = new Matcher(this, sequence, filteredResults,
        ↪ null, readAsElements);
755             matcher.AddAllPartialMatchedToResultsAndReadAsElements(res
        ↪ ults.OrderBy(x => x)); // OrderBy is a
        ↪ Hack
756             return filteredResults;
757         }
758         return new HashSet<ulong>();
759     });
760 }
761
762 // Does not work
763 public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong>
        ↪ readAsElements, params ulong[] sequence)
764 {
765     var visited = new HashSet<ulong>();
766     var results = new HashSet<ulong>();

```

```

767     var matcher = new Matcher(this, sequence, visited, x => {
        ↪ results.Add(x); return true; }, readAsElements);
768     var last = sequence.Length - 1;
769     for (var i = 0; i < last; i++)
770     {
771         PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i
        ↪ + 1]);
772     }
773     return results;
774 }
775
776 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[]
        ↪ sequence)
777 {
778     return Sync.ExecuteReadOperation(() =>
779     {
780         if (sequence.Length > 0)
781         {
782             Links.EnsureEachLinkExists(sequence);
783             //var firstElement = sequence[0];
784             //if (sequence.Length == 1)
785             //{
786                 //results.Add(firstElement);
787                 //return results;
788             //}
789             //if (sequence.Length == 2)
790             //{
791                 //var doublet = _links.SearchCore(firstElement,
        ↪ sequence[1]);
792                 //if (doublet != Doublets.Links.Null)
793                 //    results.Add(doublet);
794                 //return results;
795             //}
796             //var lastElement = sequence[sequence.Length - 1];
797             //Func<ulong, bool> handler = x =>
798             //{
799                 if (StartsWith(x, firstElement) && EndsWith(x,
        ↪ lastElement)) results.Add(x);
800                 //return true;
801             //};
802             //if (sequence.Length >= 2)
803             //    StepRight(handler, sequence[0], sequence[1]);
804             //var last = sequence.Length - 2;
805             //for (var i = 1; i < last; i++)
806             //    PartialStepRight(handler, sequence[i], sequence[i +
        ↪ 1]);
807             //if (sequence.Length >= 3)
808             //    StepLeft(handler, sequence[sequence.Length - 2],
        ↪ sequence[sequence.Length - 1]);
809             //if (sequence.Length == 1)
810             //if (sequence.Length == 1)
811             //if (sequence.Length == 1)
812             //if (sequence.Length == 1)
813             //if (sequence.Length == 1)
814             //if (sequence.Length == 1)
815             //if (sequence.Length == 1)
816             //if (sequence.Length == 1)
817             //if (sequence.Length == 1)
818             //if (sequence.Length == 1)
819             //if (sequence.Length == 1)
820             //if (sequence.Length == 1)
821             //if (sequence.Length == 1)

```

```

822     ///////////////////////////////////////////////////////////////////
823     ///////////////////////////////////////////////////////////////////   var results = new List<ulong>();
824     ///////////////////////////////////////////////////////////////////   //StepRight(results.Add, sequence[i], sequence[i
825     ///////////////////////////////////////////////////////////////////   ↪ + 1]);
826     ///////////////////////////////////////////////////////////////////   PartialStepRight(results.Add, sequence[i],
827     ///////////////////////////////////////////////////////////////////   ↪ sequence[i + 1]);
828     ///////////////////////////////////////////////////////////////////   if (results.Count > 0)
829     ///////////////////////////////////////////////////////////////////   matches.Add(results);
830     ///////////////////////////////////////////////////////////////////   else
831     ///////////////////////////////////////////////////////////////////   return results;
832     ///////////////////////////////////////////////////////////////////   if (matches.Count == 2)
833     ///////////////////////////////////////////////////////////////////   {
834     ///////////////////////////////////////////////////////////////////   var merged = new List<ulong>();
835     ///////////////////////////////////////////////////////////////////   for (var j = 0; j < matches[0].Count; j++)
836     ///////////////////////////////////////////////////////////////////   for (var k = 0; k < matches[1].Count;
837     ///////////////////////////////////////////////////////////////////   ↪ k++)
838     ///////////////////////////////////////////////////////////////////   CloseInnerConnections(merged.Add,
839     ///////////////////////////////////////////////////////////////////   ↪ matches[0][j], matches[1][k]);
840     ///////////////////////////////////////////////////////////////////   if (merged.Count > 0)
841     ///////////////////////////////////////////////////////////////////   matches = new List<List<ulong>> { merged
842     ///////////////////////////////////////////////////////////////////   ↪ };
843     ///////////////////////////////////////////////////////////////////   else
844     ///////////////////////////////////////////////////////////////////   return new List<ulong>();
845     ///////////////////////////////////////////////////////////////////   }
846     /////////////////////////////////////////////////////////////////// }
847     /////////////////////////////////////////////////////////////////// if (matches.Count > 0)
848     /////////////////////////////////////////////////////////////////// {
849     /////////////////////////////////////////////////////////////////// var usages = new HashSet<ulong>();
850     /////////////////////////////////////////////////////////////////// for (int i = 0; i < sequence.Length; i++)
851     /////////////////////////////////////////////////////////////////// {
852     /////////////////////////////////////////////////////////////////// AllUsagesCore(sequence[i], usages);
853     /////////////////////////////////////////////////////////////////// }
854     /////////////////////////////////////////////////////////////////// //for (int i = 0; i < matches[0].Count; i++)
855     /////////////////////////////////////////////////////////////////// // AllUsagesCore(matches[0][i], usages);
856     /////////////////////////////////////////////////////////////////// //usages.UnionWith(matches[0]);
857     /////////////////////////////////////////////////////////////////// return usages.ToList();
858     /////////////////////////////////////////////////////////////////// }
859     var firstLinkUsages = new HashSet<ulong>();
860     AllUsagesCore(sequence[0], firstLinkUsages);
861     firstLinkUsages.Add(sequence[0]);
862     //var previousMatchings = firstLinkUsages.ToList(); //new
863     //var previousMatchings = firstLinkUsages.ToList(); //new
864     //var previousMatchings = firstLinkUsages.ToList(); //new
865     //var previousMatchings = firstLinkUsages.ToList(); //new
866     //var previousMatchings = firstLinkUsages.ToList(); //new
867     //var previousMatchings = firstLinkUsages.ToList(); //new
868     //var previousMatchings = firstLinkUsages.ToList(); //new
869     //var previousMatchings = firstLinkUsages.ToList(); //new
870     //var previousMatchings = firstLinkUsages.ToList(); //new
871     //var previousMatchings = firstLinkUsages.ToList(); //new
872     //var previousMatchings = firstLinkUsages.ToList(); //new
873     //var previousMatchings = firstLinkUsages.ToList(); //new
874     //var previousMatchings = firstLinkUsages.ToList(); //new
875     //var previousMatchings = firstLinkUsages.ToList(); //new
876     //var previousMatchings = firstLinkUsages.ToList(); //new
877     //var previousMatchings = firstLinkUsages.ToList(); //new
878     //var previousMatchings = firstLinkUsages.ToList(); //new
879     //var previousMatchings = firstLinkUsages.ToList(); //new
880     //var previousMatchings = firstLinkUsages.ToList(); //new
881     //var previousMatchings = firstLinkUsages.ToList(); //new
882     //var previousMatchings = firstLinkUsages.ToList(); //new
883     //var previousMatchings = firstLinkUsages.ToList(); //new
884     //var previousMatchings = firstLinkUsages.ToList(); //new
885     //var previousMatchings = firstLinkUsages.ToList(); //new
886     //var previousMatchings = firstLinkUsages.ToList(); //new
887     //var previousMatchings = firstLinkUsages.ToList(); //new
888     //var previousMatchings = firstLinkUsages.ToList(); //new
889     //var previousMatchings = firstLinkUsages.ToList(); //new
890     //var previousMatchings = firstLinkUsages.ToList(); //new
891     //var previousMatchings = firstLinkUsages.ToList(); //new
892     //var previousMatchings = firstLinkUsages.ToList(); //new
893     //var previousMatchings = firstLinkUsages.ToList(); //new
894     //var previousMatchings = firstLinkUsages.ToList(); //new
895     //var previousMatchings = firstLinkUsages.ToList(); //new
896     //var previousMatchings = firstLinkUsages.ToList(); //new
897     //var previousMatchings = firstLinkUsages.ToList(); //new
898     //var previousMatchings = firstLinkUsages.ToList(); //new
899     //var previousMatchings = firstLinkUsages.ToList(); //new
900     //var previousMatchings = firstLinkUsages.ToList(); //new
901     //var previousMatchings = firstLinkUsages.ToList(); //new
902     //var previousMatchings = firstLinkUsages.ToList(); //new
903     //var previousMatchings = firstLinkUsages.ToList(); //new
904     //var previousMatchings = firstLinkUsages.ToList(); //new
905     //var previousMatchings = firstLinkUsages.ToList(); //new
906     //var previousMatchings = firstLinkUsages.ToList(); //new
907     //var previousMatchings = firstLinkUsages.ToList(); //new
908     //var previousMatchings = firstLinkUsages.ToList(); //new
909     //var previousMatchings = firstLinkUsages.ToList(); //new
910     //var previousMatchings = firstLinkUsages.ToList(); //new
911     //var previousMatchings = firstLinkUsages.ToList(); //new
912     //var previousMatchings = firstLinkUsages.ToList(); //new
913     //var previousMatchings = firstLinkUsages.ToList(); //new
914     //var previousMatchings = firstLinkUsages.ToList(); //new
915     //var previousMatchings = firstLinkUsages.ToList(); //new
916     //var previousMatchings = firstLinkUsages.ToList(); //new
917     //var previousMatchings = firstLinkUsages.ToList(); //new
918     //var previousMatchings = firstLinkUsages.ToList(); //new
919     //var previousMatchings = firstLinkUsages.ToList(); //new
920     //var previousMatchings = firstLinkUsages.ToList(); //new
921     //var previousMatchings = firstLinkUsages.ToList(); //new
922     //var previousMatchings = firstLinkUsages.ToList(); //new
923     //var previousMatchings = firstLinkUsages.ToList(); //new
924     //var previousMatchings = firstLinkUsages.ToList(); //new
925     //var previousMatchings = firstLinkUsages.ToList(); //new
926     //var previousMatchings = firstLinkUsages.ToList(); //new
927     //var previousMatchings = firstLinkUsages.ToList(); //new
928     //var previousMatchings = firstLinkUsages.ToList(); //new
929     //var previousMatchings = firstLinkUsages.ToList(); //new
930     //var previousMatchings = firstLinkUsages.ToList(); //new
931     //var previousMatchings = firstLinkUsages.ToList(); //new
932     //var previousMatchings = firstLinkUsages.ToList(); //new
933     //var previousMatchings = firstLinkUsages.ToList(); //new
934     //var previousMatchings = firstLinkUsages.ToList(); //new
935     //var previousMatchings = firstLinkUsages.ToList(); //new
936     //var previousMatchings = firstLinkUsages.ToList(); //new
937     //var previousMatchings = firstLinkUsages.ToList(); //new
938     //var previousMatchings = firstLinkUsages.ToList(); //new
939     //var previousMatchings = firstLinkUsages.ToList(); //new
940     //var previousMatchings = firstLinkUsages.ToList(); //new
941     //var previousMatchings = firstLinkUsages.ToList(); //new
942     //var previousMatchings = firstLinkUsages.ToList(); //new
943     //var previousMatchings = firstLinkUsages.ToList(); //new
944     //var previousMatchings = firstLinkUsages.ToList(); //new
945     //var previousMatchings = firstLinkUsages.ToList(); //new
946     //var previousMatchings = firstLinkUsages.ToList(); //new
947     //var previousMatchings = firstLinkUsages.ToList(); //new
948     //var previousMatchings = firstLinkUsages.ToList(); //new
949     //var previousMatchings = firstLinkUsages.ToList(); //new
950     //var previousMatchings = firstLinkUsages.ToList(); //new
951     //var previousMatchings = firstLinkUsages.ToList(); //new
952     //var previousMatchings = firstLinkUsages.ToList(); //new
953     //var previousMatchings = firstLinkUsages.ToList(); //new
954     //var previousMatchings = firstLinkUsages.ToList(); //new
955     //var previousMatchings = firstLinkUsages.ToList(); //new
956     //var previousMatchings = firstLinkUsages.ToList(); //new
957     //var previousMatchings = firstLinkUsages.ToList(); //new
958     //var previousMatchings = firstLinkUsages.ToList(); //new
959     //var previousMatchings = firstLinkUsages.ToList(); //new
960     //var previousMatchings = firstLinkUsages.ToList(); //new
961     //var previousMatchings = firstLinkUsages.ToList(); //new
962     //var previousMatchings = firstLinkUsages.ToList(); //new
963     //var previousMatchings = firstLinkUsages.ToList(); //new
964     //var previousMatchings = firstLinkUsages.ToList(); //new
965     //var previousMatchings = firstLinkUsages.ToList(); //new
966     //var previousMatchings = firstLinkUsages.ToList(); //new
967     //var previousMatchings = firstLinkUsages.ToList(); //new
968     //var previousMatchings = firstLinkUsages.ToList(); //new
969     //var previousMatchings = firstLinkUsages.ToList(); //new
970     //var previousMatchings = firstLinkUsages.ToList(); //new
971     //var previousMatchings = firstLinkUsages.ToList(); //new
972     //var previousMatchings = firstLinkUsages.ToList(); //new
973     //var previousMatchings = firstLinkUsages.ToList(); //new
974     //var previousMatchings = firstLinkUsages.ToList(); //new
975     //var previousMatchings = firstLinkUsages.ToList(); //new
976     //var previousMatchings = firstLinkUsages.ToList(); //new
977     //var previousMatchings = firstLinkUsages.ToList(); //new
978     //var previousMatchings = firstLinkUsages.ToList(); //new
979     //var previousMatchings = firstLinkUsages.ToList(); //new
980     //var previousMatchings = firstLinkUsages.ToList(); //new
981     //var previousMatchings = firstLinkUsages.ToList(); //new
982     //var previousMatchings = firstLinkUsages.ToList(); //new
983     //var previousMatchings = firstLinkUsages.ToList(); //new
984     //var previousMatchings = firstLinkUsages.ToList(); //new
985     //var previousMatchings = firstLinkUsages.ToList(); //new
986     //var previousMatchings = firstLinkUsages.ToList(); //new
987     //var previousMatchings = firstLinkUsages.ToList(); //new
988     //var previousMatchings = firstLinkUsages.ToList(); //new
989     //var previousMatchings = firstLinkUsages.ToList(); //new
990     //var previousMatchings = firstLinkUsages.ToList(); //new
991     //var previousMatchings = firstLinkUsages.ToList(); //new
992     //var previousMatchings = firstLinkUsages.ToList(); //new
993     //var previousMatchings = firstLinkUsages.ToList(); //new
994     //var previousMatchings = firstLinkUsages.ToList(); //new
995     //var previousMatchings = firstLinkUsages.ToList(); //new
996     //var previousMatchings = firstLinkUsages.ToList(); //new
997     //var previousMatchings = firstLinkUsages.ToList(); //new
998     //var previousMatchings = firstLinkUsages.ToList(); //new
999     //var previousMatchings = firstLinkUsages.ToList(); //new
1000    //var previousMatchings = firstLinkUsages.ToList(); //new

```

```

876     {
877         var usages = new HashSet<ulong>();
878         AllUsagesCore(link, usages);
879         return usages;
880     });
881 }
882
883 // При сборе всех использований (последовательностей) можно сохранять
884 // ↪ обратный путь к той связи с которой начинался поиск (STTTSSSTT),
885 // ↪ причём достаточно одного бита для хранения перехода влево или вправо
886 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
887 {
888     bool handler(ulong doublet)
889     {
890         if (usages.Add(doublet))
891         {
892             AllUsagesCore(doublet, usages);
893         }
894         return true;
895     }
896     Links.Unsync.Each(link, _constants.Any, handler);
897     Links.Unsync.Each(_constants.Any, link, handler);
898 }
899
900 public HashSet<ulong> AllBottomUsages(ulong link)
901 {
902     return Sync.ExecuteReadOperation(() =>
903     {
904         var visits = new HashSet<ulong>();
905         var usages = new HashSet<ulong>();
906         AllBottomUsagesCore(link, visits, usages);
907         return usages;
908     });
909 }
910
911 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits,
912     ↪ HashSet<ulong> usages)
913 {
914     bool handler(ulong doublet)
915     {
916         if (visits.Add(doublet))
917         {
918             AllBottomUsagesCore(doublet, visits, usages);
919         }
920         return true;
921     }
922     if (Links.Unsync.Count(_constants.Any, link) == 0)
923     {
924         usages.Add(link);
925     }
926     else
927     {
928         Links.Unsync.Each(link, _constants.Any, handler);
929         Links.Unsync.Each(_constants.Any, link, handler);
930     }
931 }
932
933 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
934 {
935     if (Options.UseSequenceMarker)
936     {
937         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter
938         {
939             ↪ ter<ulong>(Links, Options.MarkedSequenceMatcher,
940             ↪ symbol);
941         return counter.Count();
942     }
943 }

```



```

937     }
938     else
939     {
940         var counter = new
            ↳ TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
            ↳ symbol);
941         return counter.Count();
942     }
943 }
944
945 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages,
    ↳ Func<ulong, bool> outerHandler)
946 {
947     bool handler(ulong doublet)
948     {
949         if (usages.Add(doublet))
950         {
951             if (!outerHandler(doublet))
952             {
953                 return false;
954             }
955             if (!AllUsagesCore1(doublet, usages, outerHandler))
956             {
957                 return false;
958             }
959         }
960         return true;
961     }
962     return Links.Unsync.Each(link, _constants.Any, handler)
        && Links.Unsync.Each(_constants.Any, link, handler);
963 }
964
965 public void CalculateAllUsages(ulong[] totals)
966 {
967     var calculator = new AllUsagesCalculator(Links, totals);
968     calculator.Calculate();
969 }
970
971 public void CalculateAllUsages2(ulong[] totals)
972 {
973     var calculator = new AllUsagesCalculator2(Links, totals);
974     calculator.Calculate();
975 }
976
977 private class AllUsagesCalculator
978 {
979     private readonly SynchronizedLinks<ulong> _links;
980     private readonly ulong[] _totals;
981
982     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[]
    ↳ totals)
983     {
984         _links = links;
985         _totals = totals;
986     }
987
988     public void Calculate() => _links.Each(_constants.Any,
    ↳ _constants.Any, CalculateCore);
989
990     private bool CalculateCore(ulong link)
991     {
992         if (_totals[link] == 0)
993         {
994             var total = 1UL;
995             _totals[link] = total;
996             var visitedChildren = new HashSet<ulong>();

```

```

998         bool linkCalculator(ulong child)
999         {
1000             if (link != child && visitedChildren.Add(child))
1001             {
1002                 total += _totals[child] == 0 ? 1 : _totals[child];
1003             }
1004             return true;
1005         }
1006         _links.Unsync.Each(link, _constants.Any, linkCalculator);
1007         _links.Unsync.Each(_constants.Any, link, linkCalculator);
1008         _totals[link] = total;
1009     }
1010     return true;
1011 }
1012
1013 private class AllUsagesCalculator2
1014 {
1015     private readonly SynchronizedLinks<ulong> _links;
1016     private readonly ulong[] _totals;
1017
1018     public AllUsagesCalculator2(SynchronizedLinks<ulong> links,
    ↳ ulong[] totals)
1019     {
1020         _links = links;
1021         _totals = totals;
1022     }
1023
1024     public void Calculate() => _links.Each(_constants.Any,
    ↳ _constants.Any, CalculateCore);
1025
1026     private bool IsElement(ulong link)
1027     {
1028         // _linksInSequence.Contains(link) ||
1029         return _links.Unsync.GetTarget(link) == link ||
            ↳ _links.Unsync.GetSource(link) == link;
1030     }
1031
1032     private bool CalculateCore(ulong link)
1033     {
1034         // TODO: Проработать защиту от заикливания
1035         // Основано на SequenceWalker.WalkLeft
1036         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1037         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1038         Func<ulong, bool> isElement = IsElement;
1039         void visitLeaf(ulong parent)
1040         {
1041             if (link != parent)
1042             {
1043                 _totals[parent]++;
1044             }
1045         }
1046         void visitNode(ulong parent)
1047         {
1048             if (link != parent)
1049             {
1050                 _totals[parent]++;
1051             }
1052         }
1053         var stack = new Stack();
1054         var element = link;
1055         if (isElement(element))
1056         {
1057             visitLeaf(element);
1058         }
1059         else
1060

```

```

1061 {
1062     while (true)
1063     {
1064         if (isElement(element))
1065         {
1066             if (stack.Count == 0)
1067             {
1068                 break;
1069             }
1070             element = stack.Pop();
1071             var source = getSource(element);
1072             var target = getTarget(element);
1073             // Обработка элемента
1074             if (isElement(target))
1075             {
1076                 visitLeaf(target);
1077             }
1078             if (isElement(source))
1079             {
1080                 visitLeaf(source);
1081             }
1082             element = source;
1083         }
1084         else
1085         {
1086             stack.Push(element);
1087             visitNode(element);
1088             element = getTarget(element);
1089         }
1090     }
1091     _totals[link]++;
1092     return true;
1093 }
1094 }
1095 }
1096
1097 private class AllUsagesCollector
1098 {
1099     private readonly ILinks<ulong> _links;
1100     private readonly HashSet<ulong> _usages;
1101
1102     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong>
1103     ↪ usages)
1104     {
1105         _links = links;
1106         _usages = usages;
1107     }
1108
1109     public bool Collect(ulong link)
1110     {
1111         if (_usages.Add(link))
1112         {
1113             _links.Each(link, _constants.Any, Collect);
1114             _links.Each(_constants.Any, link, Collect);
1115         }
1116         return true;
1117     }
1118 }
1119
1120 private class AllUsagesCollector1
1121 {
1122     private readonly ILinks<ulong> _links;
1123     private readonly HashSet<ulong> _usages;
1124     private readonly ulong _continue;
1125
1126     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong>
1127     ↪ usages)

```

```

1126 {
1127     _links = links;
1128     _usages = usages;
1129     _continue = _links.Constants.Continue;
1130 }
1131
1132 public ulong Collect(IList<ulong> link)
1133 {
1134     var linkIndex = _links.GetIndex(link);
1135     if (_usages.Add(linkIndex))
1136     {
1137         _links.Each(Collect, _constants.Any, linkIndex);
1138     }
1139     return _continue;
1140 }
1141 }
1142
1143 private class AllUsagesCollector2
1144 {
1145     private readonly ILinks<ulong> _links;
1146     private readonly BitString _usages;
1147
1148     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1149     {
1150         _links = links;
1151         _usages = usages;
1152     }
1153
1154     public bool Collect(ulong link)
1155     {
1156         if (_usages.Add((long)link))
1157         {
1158             _links.Each(link, _constants.Any, Collect);
1159             _links.Each(_constants.Any, link, Collect);
1160         }
1161         return true;
1162     }
1163 }
1164
1165 private class AllUsagesIntersectingCollector
1166 {
1167     private readonly SynchronizedLinks<ulong> _links;
1168     private readonly HashSet<ulong> _intersectWith;
1169     private readonly HashSet<ulong> _usages;
1170     private readonly HashSet<ulong> _enter;
1171
1172     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong>
1173     ↪ links, HashSet<ulong> intersectWith, HashSet<ulong> usages)
1174     {
1175         _links = links;
1176         _intersectWith = intersectWith;
1177         _usages = usages;
1178         _enter = new HashSet<ulong>(); // защита от зацикливания
1179     }
1180
1181     public bool Collect(ulong link)
1182     {
1183         if (_enter.Add(link))
1184         {
1185             if (_intersectWith.Contains(link))
1186             {
1187                 _usages.Add(link);
1188             }
1189             _links.Unsync.Each(link, _constants.Any, Collect);
1190             _links.Unsync.Each(_constants.Any, link, Collect);
1191         }
1192         return true;
1193     }

```

```

1192     }
1193 }
1194
1195 private void CloseInnerConnections(Action<ulong> handler, ulong left,
    ↳ ulong right)
1196 {
1197     TryStepLeftUp(handler, left, right);
1198     TryStepRightUp(handler, right, left);
1199 }
1200
1201 private void AllCloseConnections(Action<ulong> handler, ulong left,
    ↳ ulong right)
1202 {
1203     // Direct
1204     if (left == right)
1205     {
1206         handler(left);
1207     }
1208     var doublet = Links.Unsync.SearchOrDefault(left, right);
1209     if (doublet != _constants.Null)
1210     {
1211         handler(doublet);
1212     }
1213     // Inner
1214     CloseInnerConnections(handler, left, right);
1215     // Outer
1216     StepLeft(handler, left, right);
1217     StepRight(handler, left, right);
1218     PartialStepRight(handler, left, right);
1219     PartialStepLeft(handler, left, right);
1220 }
1221
1222 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[]
    ↳ sequence, HashSet<ulong> previousMatchings, long startAt)
1223 {
1224     if (startAt >= sequence.Length) // ?
1225     {
1226         return previousMatchings;
1227     }
1228     var secondLinkUsages = new HashSet<ulong>();
1229     AllUsagesCore(sequence[startAt], secondLinkUsages);
1230     secondLinkUsages.Add(sequence[startAt]);
1231     var matchings = new HashSet<ulong>();
1232     //for (var i = 0; i < previousMatchings.Count; i++)
1233     foreach (var secondLinkUsage in secondLinkUsages)
1234     {
1235         foreach (var previousMatching in previousMatchings)
1236         {
1237             //AllCloseConnections(matchings.AddAndReturnVoid,
    ↳ previousMatching, secondLinkUsage);
1238             StepRight(matchings.AddAndReturnVoid, previousMatching,
    ↳ secondLinkUsage);
1239             TryStepRightUp(matchings.AddAndReturnVoid,
    ↳ secondLinkUsage, previousMatching);
1240             //PartialStepRight(matchings.AddAndReturnVoid,
    ↳ secondLinkUsage, sequence[startAt]); // почему-то эта
    ↳ ошибочная запись приводит к желаемым результатам.
1241             PartialStepRight(matchings.AddAndReturnVoid,
    ↳ previousMatching, secondLinkUsage);
1242         }
1243     }
1244     if (matchings.Count == 0)
1245     {
1246         return matchings;
1247     }

```

```

1248     return GetAllPartiallyMatchingSequencesCore(sequence, matchings,
    ↳ startAt + 1); // ??
1249 }
1250
1251 private static void
    ↳ EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
    ↳ links, params ulong[] sequence)
1252 {
1253     if (sequence == null)
1254     {
1255         return;
1256     }
1257     for (var i = 0; i < sequence.Length; i++)
1258     {
1259         if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany
    ↳ && !links.Exists(sequence[i]))
1260         {
1261             throw new
    ↳ ArgumentLinkDoesNotExistException<ulong>(sequence[i],
    ↳ $"patternSequence[{i}]");
1262         }
1263     }
1264 }
1265
1266 // Pattern Matching -> Key To Triggers
1267 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1268 {
1269     return Sync.ExecuteReadOperation(() =>
1270     {
1271         patternSequence = Simplify(patternSequence);
1272         if (patternSequence.Length > 0)
1273         {
1274             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links,
    ↳ patternSequence);
1275             var uniqueSequenceElements = new HashSet<ulong>();
1276             for (var i = 0; i < patternSequence.Length; i++)
1277             {
1278                 if (patternSequence[i] != _constants.Any &&
    ↳ patternSequence[i] != ZeroOrMany)
1279                 {
1280                     uniqueSequenceElements.Add(patternSequence[i]);
1281                 }
1282             }
1283             var results = new HashSet<ulong>();
1284             foreach (var uniqueSequenceElement in
    ↳ uniqueSequenceElements)
1285             {
1286                 AllUsagesCore(uniqueSequenceElement, results);
1287             }
1288             var filteredResults = new HashSet<ulong>();
1289             var matcher = new PatternMatcher(this, patternSequence,
    ↳ filteredResults);
1290             matcher.AddAllPatternMatchedToResults(results);
1291             return filteredResults;
1292         }
1293         return new HashSet<ulong>();
1294     });
1295 }
1296
1297 // Найти все возможные связи между указанным списком связей.
1298 // Находит связи между всеми указанными связями в любом порядке.
1299 // TODO: решить что делать с повторами (когда одни и те же элементы
    ↳ встречаются несколько раз в последовательности)
1300 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)

```

```

1301 {
1302     return Sync.ExecuteReadOperation(() =>
1303     {
1304         var results = new HashSet<ulong>();
1305         if (linksToConnect.Length > 0)
1306         {
1307             Links.EnsureEachLinkExists(linksToConnect);
1308             AllUsagesCore(linksToConnect[0], results);
1309             for (var i = 1; i < linksToConnect.Length; i++)
1310             {
1311                 var next = new HashSet<ulong>();
1312                 AllUsagesCore(linksToConnect[i], next);
1313                 results.IntersectWith(next);
1314             }
1315         }
1316         return results;
1317     });
1318 }
1319 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1320 {
1321     return Sync.ExecuteReadOperation(() =>
1322     {
1323         var results = new HashSet<ulong>();
1324         if (linksToConnect.Length > 0)
1325         {
1326             Links.EnsureEachLinkExists(linksToConnect);
1327             var collector1 = new AllUsagesCollector(Links.Unsync,
1328             ↪ results);
1329             collector1.Collect(linksToConnect[0]);
1330             var next = new HashSet<ulong>();
1331             for (var i = 1; i < linksToConnect.Length; i++)
1332             {
1333                 var collector = new AllUsagesCollector(Links.Unsync,
1334                 ↪ next);
1335                 collector.Collect(linksToConnect[i]);
1336                 results.IntersectWith(next);
1337                 next.Clear();
1338             }
1339             return results;
1340         });
1341 }
1342 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1343 {
1344     return Sync.ExecuteReadOperation(() =>
1345     {
1346         var results = new HashSet<ulong>();
1347         if (linksToConnect.Length > 0)
1348         {
1349             Links.EnsureEachLinkExists(linksToConnect);
1350             var collector1 = new AllUsagesCollector(Links, results);
1351             collector1.Collect(linksToConnect[0]);
1352             //AllUsagesCore(linksToConnect[0], results);
1353             for (var i = 1; i < linksToConnect.Length; i++)
1354             {
1355                 var next = new HashSet<ulong>();
1356                 var collector = new
1357                 ↪ AllUsagesIntersectingCollector(Links, results,
1358                 ↪ next);
1359                 collector.Collect(linksToConnect[i]);
1360                 //AllUsagesCore(linksToConnect[i], next);
1361                 //results.IntersectWith(next);
1362                 results = next;

```

```

1362     }
1363     }
1364     return results;
1365 });
1366 }
1367
1368 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1369 {
1370     return Sync.ExecuteReadOperation(() =>
1371     {
1372         var results = new BitString((long)Links.Unsync.Count() + 1);
1373         ↪ // new BitArray((int)_links.Total + 1);
1374         if (linksToConnect.Length > 0)
1375         {
1376             Links.EnsureEachLinkExists(linksToConnect);
1377             var collector1 = new AllUsagesCollector2(Links.Unsync,
1378             ↪ results);
1379             collector1.Collect(linksToConnect[0]);
1380             for (var i = 1; i < linksToConnect.Length; i++)
1381             {
1382                 var next = new BitString((long)Links.Unsync.Count() +
1383                 ↪ 1); //new BitArray((int)_links.Total + 1);
1384                 var collector = new AllUsagesCollector2(Links.Unsync,
1385                 ↪ next);
1386                 collector.Collect(linksToConnect[i]);
1387                 results = results.And(next);
1388             }
1389             return results.GetSetUInt64Indices();
1390         });
1391 }
1392
1393 private static ulong[] Simplify(ulong[] sequence)
1394 {
1395     // Считаем новый размер последовательности
1396     long newLength = 0;
1397     var zeroOrManyStepped = false;
1398     for (var i = 0; i < sequence.Length; i++)
1399     {
1400         if (sequence[i] == ZeroOrMany)
1401         {
1402             if (zeroOrManyStepped)
1403             {
1404                 continue;
1405             }
1406             zeroOrManyStepped = true;
1407         }
1408         else
1409         {
1410             //if (zeroOrManyStepped) Is it efficient?
1411             zeroOrManyStepped = false;
1412             newLength++;
1413         }
1414     }
1415     // Строим новую последовательность
1416     zeroOrManyStepped = false;
1417     var newSequence = new ulong[newLength];
1418     long j = 0;
1419     for (var i = 0; i < sequence.Length; i++)
1420     {
1421         //var current = zeroOrManyStepped;
1422         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1423         //if (current && zeroOrManyStepped)
1424         //    continue;
1425         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1426         //if (zeroOrManyStepped && newZeroOrManyStepped)

```

```

1424     // continue;
1425     //zeroOrManyStepped = newZeroOrManyStepped;
1426     if (sequence[i] == ZeroOrMany)
1427     {
1428         if (zeroOrManyStepped)
1429         {
1430             continue;
1431         }
1432         zeroOrManyStepped = true;
1433     }
1434     else
1435     {
1436         //if (zeroOrManyStepped) Is it efficient?
1437         zeroOrManyStepped = false;
1438     }
1439     newSequence[j++] = sequence[i];
1440 }
1441 return newSequence;
1442 }
1443
1444 public static void TestSimplify()
1445 {
1446     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4,
        ↳ ZeroOrMany, ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany,
        ↳ ZeroOrMany };
1447     var simplifiedSequence = Simplify(sequence);
1448 }
1449
1450 public List<ulong> GetSimilarSequences() => new List<ulong>();
1451
1452 public void Prediction()
1453 {
1454     //_links
1455     //_sequences
1456 }
1457
1458 #region From Triplets
1459
1460 //public static void DeleteSequence(Link sequence)
1461 //{
1462 //}
1463
1464 public List<ulong> CollectMatchingSequences(ulong[] links)
1465 {
1466     if (links.Length == 1)
1467     {
1468         throw new Exception("Подпоследовательности с одним элементом
        ↳ не поддерживаются.");
1469     }
1470     var leftBound = 0;
1471     var rightBound = links.Length - 1;
1472     var left = links[leftBound++];
1473     var right = links[rightBound--];
1474     var results = new List<ulong>();
1475     CollectMatchingSequences(left, leftBound, links, right,
        ↳ rightBound, ref results);
1476     return results;
1477 }
1478
1479 private void CollectMatchingSequences(ulong leftLink, int leftBound,
        ↳ ulong[] middleLinks, ulong rightLink, int rightBound, ref
        ↳ List<ulong> results)
1480 {
1481     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1482     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1483     if (leftLinkTotalReferers <= rightLinkTotalReferers)

```

```

1484 {
1485     var nextLeftLink = middleLinks[leftBound];
1486     var elements = GetRightElements(leftLink, nextLeftLink);
1487     if (leftBound <= rightBound)
1488     {
1489         for (var i = elements.Length - 1; i >= 0; i--)
1490         {
1491             var element = elements[i];
1492             if (element != 0)
1493             {
1494                 CollectMatchingSequences(element, leftBound + 1,
                    ↳ middleLinks, rightLink, rightBound, ref
                    ↳ results);
1495             }
1496         }
1497     }
1498     else
1499     {
1500         for (var i = elements.Length - 1; i >= 0; i--)
1501         {
1502             var element = elements[i];
1503             if (element != 0)
1504             {
1505                 results.Add(element);
1506             }
1507         }
1508     }
1509 }
1510 else
1511 {
1512     var nextRightLink = middleLinks[rightBound];
1513     var elements = GetLeftElements(rightLink, nextRightLink);
1514     if (leftBound <= rightBound)
1515     {
1516         for (var i = elements.Length - 1; i >= 0; i--)
1517         {
1518             var element = elements[i];
1519             if (element != 0)
1520             {
1521                 CollectMatchingSequences(leftLink, leftBound,
                    ↳ middleLinks, elements[i], rightBound - 1, ref
                    ↳ results);
1522             }
1523         }
1524     }
1525     else
1526     {
1527         for (var i = elements.Length - 1; i >= 0; i--)
1528         {
1529             var element = elements[i];
1530             if (element != 0)
1531             {
1532                 results.Add(element);
1533             }
1534         }
1535     }
1536 }
1537 }
1538
1539 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1540 {
1541     var result = new ulong[5];
1542     TryStepRight(startLink, rightLink, result, 0);
1543     Links.Each(_constants.Any, startLink, couple =>
1544     {

```

```

1545     if (couple != startLink)
1546     {
1547         if (TryStepRight(couple, rightLink, result, 2))
1548         {
1549             return false;
1550         }
1551     }
1552     return true;
1553 });
1554 if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1555 {
1556     result[4] = startLink;
1557 }
1558 return result;
1559 }
1560
1561 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[]
↪ result, int offset)
1562 {
1563     var added = 0;
1564     Links.Each(startLink, _constants.Any, couple =>
1565     {
1566         if (couple != startLink)
1567         {
1568             var coupleTarget = Links.GetTarget(couple);
1569             if (coupleTarget == rightLink)
1570             {
1571                 result[offset] = couple;
1572                 if (++added == 2)
1573                 {
1574                     return false;
1575                 }
1576             }
1577             else if (Links.GetSource(coupleTarget) == rightLink) //
↪ coupleTarget.Linker == Net.And &&
1578             {
1579                 result[offset + 1] = couple;
1580                 if (++added == 2)
1581                 {
1582                     return false;
1583                 }
1584             }
1585         }
1586         return true;
1587     });
1588     return added > 0;
1589 }
1590
1591 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1592 {
1593     var result = new ulong[5];
1594     TryStepLeft(startLink, leftLink, result, 0);
1595     Links.Each(startLink, _constants.Any, couple =>
1596     {
1597         if (couple != startLink)
1598         {
1599             if (TryStepLeft(couple, leftLink, result, 2))
1600             {
1601                 return false;
1602             }
1603         }
1604         return true;
1605     });
1606     if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1607     {
1608         result[4] = leftLink;

```

```

1609     }
1610     return result;
1611 }
1612
1613 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[]
↪ result, int offset)
1614 {
1615     var added = 0;
1616     Links.Each(_constants.Any, startLink, couple =>
1617     {
1618         if (couple != startLink)
1619         {
1620             var coupleSource = Links.GetSource(couple);
1621             if (coupleSource == leftLink)
1622             {
1623                 result[offset] = couple;
1624                 if (++added == 2)
1625                 {
1626                     return false;
1627                 }
1628             }
1629             else if (Links.GetTarget(coupleSource) == leftLink) //
↪ coupleSource.Linker == Net.And &&
1630             {
1631                 result[offset + 1] = couple;
1632                 if (++added == 2)
1633                 {
1634                     return false;
1635                 }
1636             }
1637         }
1638         return true;
1639     });
1640     return added > 0;
1641 }
1642
1643 #endregion
1644
1645 #region Walkers
1646
1647 public class PatternMatcher : RightSequenceWalker<ulong>
1648 {
1649     private readonly Sequences _sequences;
1650     private readonly ulong[] _patternSequence;
1651     private readonly HashSet<LinkIndex> _linksInSequence;
1652     private readonly HashSet<LinkIndex> _results;
1653
1654     #region Pattern Match
1655
1656     enum PatternBlockType
1657     {
1658         Undefined,
1659         Gap,
1660         Elements
1661     }
1662
1663     struct PatternBlock
1664     {
1665         public PatternBlockType Type;
1666         public long Start;
1667         public long Stop;
1668     }
1669
1670     private readonly List<PatternBlock> _pattern;
1671     private int _patternPosition;
1672     private long _sequencePosition;
1673
1674     #endregion

```

```

1675 public PatternMatcher(Sequences sequences, LinkIndex[]
1676     ↳ patternSequence, HashSet<LinkIndex> results)
1677     : base(sequences.Links.Unsync)
1678 {
1679     _sequences = sequences;
1680     _patternSequence = patternSequence;
1681     _linksInSequence = new
1682         ↳ HashSet<LinkIndex>(patternSequence.Where(x => x !=
1683         ↳ _constants.Any && x != ZeroOrMany));
1684     _results = results;
1685     _pattern = CreateDetailedPattern();
1686
1687 protected override bool IsElement(IList<ulong> link) =>
1688     ↳ _linksInSequence.Contains(Links.GetIndex(link)) ||
1689     ↳ base.IsElement(link);
1690
1691 public bool PatternMatch(LinkIndex sequenceToMatch)
1692 {
1693     _patternPosition = 0;
1694     _sequencePosition = 0;
1695     foreach (var part in Walk(sequenceToMatch))
1696     {
1697         if (!PatternMatchCore(Links.GetIndex(part)))
1698         {
1699             break;
1700         }
1701     }
1702     return _patternPosition == _pattern.Count || (_patternPosition
1703     ↳ == _pattern.Count - 1 && _pattern[_patternPosition].Start
1704     ↳ == 0);
1705 }
1706
1707 private List<PatternBlock> CreateDetailedPattern()
1708 {
1709     var pattern = new List<PatternBlock>();
1710     var patternBlock = new PatternBlock();
1711     for (var i = 0; i < _patternSequence.Length; i++)
1712     {
1713         if (patternBlock.Type == PatternBlockType.Undefined)
1714         {
1715             if (_patternSequence[i] == _constants.Any)
1716             {
1717                 patternBlock.Type = PatternBlockType.Gap;
1718                 patternBlock.Start = 1;
1719                 patternBlock.Stop = 1;
1720             }
1721             else if (_patternSequence[i] == ZeroOrMany)
1722             {
1723                 patternBlock.Type = PatternBlockType.Gap;
1724                 patternBlock.Start = 0;
1725                 patternBlock.Stop = long.MaxValue;
1726             }
1727             else
1728             {
1729                 patternBlock.Type = PatternBlockType.Elements;
1730                 patternBlock.Start = i;
1731                 patternBlock.Stop = i;
1732             }
1733         }
1734         else if (patternBlock.Type == PatternBlockType.Elements)
1735         {
1736             if (_patternSequence[i] == _constants.Any)
1737             {
1738                 pattern.Add(patternBlock);

```

```

1734         patternBlock = new PatternBlock
1735         {
1736             Type = PatternBlockType.Gap,
1737             Start = 1,
1738             Stop = 1
1739         };
1740     }
1741     else if (_patternSequence[i] == ZeroOrMany)
1742     {
1743         pattern.Add(patternBlock);
1744         patternBlock = new PatternBlock
1745         {
1746             Type = PatternBlockType.Gap,
1747             Start = 0,
1748             Stop = long.MaxValue
1749         };
1750     }
1751     else
1752     {
1753         patternBlock.Stop = i;
1754     }
1755 }
1756 else // patternBlock.Type == PatternBlockType.Gap
1757 {
1758     if (_patternSequence[i] == _constants.Any)
1759     {
1760         patternBlock.Start++;
1761         if (patternBlock.Stop < patternBlock.Start)
1762         {
1763             patternBlock.Stop = patternBlock.Start;
1764         }
1765     }
1766     else if (_patternSequence[i] == ZeroOrMany)
1767     {
1768         patternBlock.Stop = long.MaxValue;
1769     }
1770     else
1771     {
1772         pattern.Add(patternBlock);
1773         patternBlock = new PatternBlock
1774         {
1775             Type = PatternBlockType.Elements,
1776             Start = i,
1777             Stop = i
1778         };
1779     }
1780 }
1781 }
1782 if (patternBlock.Type != PatternBlockType.Undefined)
1783 {
1784     pattern.Add(patternBlock);
1785 }
1786 return pattern;
1787 }
1788
1789 /* match: search for regexp anywhere in text */
1790 int match(char* regexp, char* text)
1791 {
1792     do
1793     {
1794         while (*text++ != '\0');
1795         return 0;
1796     }
1797 }
1798
1799 /* matchhere: search for regexp at beginning of text */
1800 int matchhere(char* regexp, char* text)
1801 {

```

```

1801 // if (regexp[0] == '\0')
1802 // return 1;
1803 // if (regexp[1] == '*')
1804 // return matchstar(regexp[0], regexp + 2, text);
1805 // if (regexp[0] == '$' && regexp[1] == '\0')
1806 // return *text == '\0';
1807 // if (*text != '\0' && (regexp[0] == '.' || regexp[0] ==
→ *text))
1808 // return matchhere(regexp + 1, text + 1);
1809 // return 0;
1810 //}
1811
1812 /** matchstar: search for c*regexp at beginning of text */
1813 //int matchstar(int c, char* regexp, char* text)
1814 //{
1815 // do
1816 // { /* a * matches zero or more instances */
1817 // if (matchhere(regexp, text))
1818 // return 1;
1819 // } while (*text != '\0' && (*text++ == c || c == '.'));
1820 // return 0;
1821 //}
1822
1823 //private void GetNextPatternElement(out LinkIndex element, out
→ long mininumGap, out long maximumGap)
1824 //{
1825 // mininumGap = 0;
1826 // maximumGap = 0;
1827 // element = 0;
1828 // for (; _patternPosition < _patternSequence.Length;
→ _patternPosition++)
1829 // {
1830 // if (_patternSequence[_patternPosition] ==
→ Doublets.Links.Null)
1831 // mininumGap++;
1832 // else if (_patternSequence[_patternPosition] ==
→ ZeroOrMany)
1833 // maximumGap = long.MaxValue;
1834 // else
1835 // break;
1836 // }
1837
1838 // if (maximumGap < mininumGap)
1839 // maximumGap = mininumGap;
1840 //}
1841
1842 private bool PatternMatchCore(LinkIndex element)
1843 {
1844 if (_patternPosition >= _pattern.Count)
1845 {
1846 _patternPosition = -2;
1847 return false;
1848 }
1849 var currentPatternBlock = _pattern[_patternPosition];
1850 if (currentPatternBlock.Type == PatternBlockType.Gap)
1851 {
1852 //var currentMatchingBlockLength = (_sequencePosition -
→ _lastMatchedBlockPosition);
1853 if (_sequencePosition < currentPatternBlock.Start)
1854 {
1855 _sequencePosition++;
1856 return true; // Двигаемся дальше
1857 }
1858 // Это последний блок
1859 if (_pattern.Count == _patternPosition + 1)

```

```

1860 {
1861 _patternPosition++;
1862 _sequencePosition = 0;
1863 return false; // Полное соответствие
1864 }
1865 else
1866 {
1867 if (_sequencePosition > currentPatternBlock.Stop)
1868 {
1869 return false; // Соответствие невозможно
1870 }
1871 var nextPatternBlock = _pattern[_patternPosition + 1];
1872 if (_patternSequence[nextPatternBlock.Start] ==
→ element)
1873 {
1874 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1875 {
1876 _patternPosition++;
1877 _sequencePosition = 1;
1878 }
1879 else
1880 {
1881 _patternPosition += 2;
1882 _sequencePosition = 0;
1883 }
1884 }
1885 }
1886 }
1887 else // currentPatternBlock.Type == PatternBlockType.Elements
1888 {
1889 var patternElementPosition = currentPatternBlock.Start +
→ _sequencePosition;
1890 if (_patternSequence[patternElementPosition] != element)
1891 {
1892 return false; // Соответствие невозможно
1893 }
1894 if (patternElementPosition == currentPatternBlock.Stop)
1895 {
1896 _patternPosition++;
1897 _sequencePosition = 0;
1898 }
1899 else
1900 {
1901 _sequencePosition++;
1902 }
1903 }
1904 return true;
1905 //if (_patternSequence[_patternPosition] != element)
1906 // return false;
1907 //else
1908 //{
1909 // _sequencePosition++;
1910 // _patternPosition++;
1911 // return true;
1912 //}
1913 //if (_filterPosition == _patternSequence.Length)
1914 //{
1915 // _filterPosition = -2; // Длиннее чем нужно
1916 // return false;
1917 //}
1918 //if (element != _patternSequence[_filterPosition])
1919 //{
1920 // _filterPosition = -1;
1921 // return false; // Начинается иначе
1922 //}
1923

```



```

1924         //_filterPosition++;
1925         //if (_filterPosition == (_patternSequence.Length - 1))
1926         //    return false;
1927         //if (_filterPosition >= 0)
1928         //{
1929             if (element == _patternSequence[_filterPosition + 1])
1930             //    _filterPosition++;
1931             //    else
1932             //        return false;
1933             //}
1934         //if (_filterPosition < 0)
1935         //{
1936             if (element == _patternSequence[0])
1937             //    _filterPosition = 0;
1938             //}
1939     }
1940
1941     public void AddAllPatternMatchedToResults(IEnumerable<ulong>
    ↪ sequencesToMatch)
1942     {
1943         foreach (var sequenceToMatch in sequencesToMatch)
1944         {
1945             if (PatternMatch(sequenceToMatch))
1946             {
1947                 _results.Add(sequenceToMatch);
1948             }
1949         }
1950     }
1951 }
1952 #endregion
1953 }
1954 }
1955 }

```

./Sequences/Sequences.Experiments.ReadSequence.cs

```

1  // #define USEARRAYPOOL
2  using System;
3  using System.Runtime.CompilerServices;
4  #if USEARRAYPOOL
5  using Platform.Collections;
6  #endif
7
8  namespace Platform.Data.Doublets.Sequences
9  {
10     partial class Sequences
11     {
12         public ulong[] ReadSequenceCore(ulong sequence, Func<ulong, bool>
    ↪ isElement)
13         {
14             var links = Links.Unsync;
15             var length = 1;
16             var array = new ulong[length];
17             array[0] = sequence;
18
19             if (isElement(sequence))
20             {
21                 return array;
22             }
23
24             bool hasElements;
25             do
26             {
27                 length *= 2;
28             #if USEARRAYPOOL
29                 var nextArray = ArrayPool.Allocate<ulong>(length);
30             #else
31                 var nextArray = new ulong[length];

```

```

32     #endif
33
34     hasElements = false;
35     for (var i = 0; i < array.Length; i++)
36     {
37         var candidate = array[i];
38         if (candidate == 0)
39         {
40             continue;
41         }
42         var doubletOffset = i * 2;
43         if (isElement(candidate))
44         {
45             nextArray[doubletOffset] = candidate;
46         }
47         else
48         {
49             var link = links.GetLink(candidate);
50             var linkSource = links.GetSource(link);
51             var linkTarget = links.GetTarget(link);
52             nextArray[doubletOffset] = linkSource;
53             nextArray[doubletOffset + 1] = linkTarget;
54             if (!hasElements)
55             {
56                 hasElements = !(isElement(linkSource) &&
    ↪ isElement(linkTarget));
57             }
58         }
59     }
60     #if USEARRAYPOOL
61     if (array.Length > 1)
62     {
63         ArrayPool.Free(array);
64     }
65     #endif
66     array = nextArray;
67
68     while (hasElements);
69     var filledElementsCount = CountFilledElements(array);
70     if (filledElementsCount == array.Length)
71     {
72         return array;
73     }
74     else
75     {
76         return CopyFilledElements(array, filledElementsCount);
77     }
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 private static ulong[] CopyFilledElements(ulong[] array, int
    ↪ filledElementsCount)
82 {
83     var finalArray = new ulong[filledElementsCount];
84     for (int i = 0, j = 0; i < array.Length; i++)
85     {
86         if (array[i] > 0)
87         {
88             finalArray[j] = array[i];
89             j++;
90         }
91     }
92     #if USEARRAYPOOL
93     ArrayPool.Free(array);
94     #endif
95     return finalArray;
96 }

```

```

96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     private static int CountFilledElements(ulong[] array)
98     {
99         var count = 0;
100         for (var i = 0; i < array.Length; i++)
101         {
102             if (array[i] > 0)
103             {
104                 count++;
105             }
106         }
107         return count;
108     }
109 }
110 }
111 }

```

./Sequences/SequencesExtensions.cs

```

1  using Platform.Data.Sequences;
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets.Sequences
5  {
6      public static class SequencesExtensions
7      {
8          public static TLink Create<TLink>(this ISequences<TLink> sequences,
9              ↳ IList<TLink[]> groupedSequence)
10         {
11             var finalSequence = new TLink[groupedSequence.Count];
12             for (var i = 0; i < finalSequence.Length; i++)
13             {
14                 var part = groupedSequence[i];
15                 finalSequence[i] = part.Length == 1 ? part[0] :
16                     ↳ sequences.Create(part);
17             }
18             return sequences.Create(finalSequence);
19         }
20     }
21 }

```

./Sequences/SequencesIndexer.cs

```

1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Sequences
4  {
5      public class SequencesIndexer<TLink>
6      {
7          private static readonly EqualityComparer<TLink> _equalityComparer =
8              ↳ EqualityComparer<TLink>.Default;
9
10         private readonly ISynchronizedLinks<TLink> _links;
11         private readonly TLink _null;
12
13         public SequencesIndexer(ISynchronizedLinks<TLink> links)
14         {
15             _links = links;
16             _null = _links.Constants.Null;
17         }
18
19         /// <summary>
20         /// Индексирует последовательность глобально, и возвращает значение,
21         /// определяющие была ли запрошенная последовательность
22         /// проиндексирована ранее.
23         /// </summary>
24         /// <param name="sequence">Последовательность для индексации.</param>
25         /// <returns>

```

```

24     /// True если последовательность уже была проиндексирована ранее и
25     /// False если последовательность была проиндексирована только что.
26     /// </returns>
27     public bool Index(TLink[] sequence)
28     {
29         var indexed = true;
30         var i = sequence.Length;
31         while (--i >= 1 && (indexed =
32             ↳ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i -
33             ↳ 1], sequence[i]), _null))) { }
34         for (; i >= 1; i--)
35         {
36             _links.GetOrCreate(sequence[i - 1], sequence[i]);
37         }
38         return indexed;
39     }
40
41     public bool BulkIndex(TLink[] sequence)
42     {
43         var indexed = true;
44         var i = sequence.Length;
45         var links = _links.Unsync;
46         _links.SyncRoot.ExecuteReadOperation(() =>
47         {
48             while (--i >= 1 && (indexed =
49                 ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i
50                 ↳ - 1], sequence[i]), _null))) { }
51         });
52         if (indexed == false)
53         {
54             _links.SyncRoot.ExecuteWriteOperation(() =>
55             {
56                 for (; i >= 1; i--)
57                 {
58                     links.GetOrCreate(sequence[i - 1], sequence[i]);
59                 }
60             });
61         }
62         return indexed;
63     }
64
65     public bool BulkIndexUnsync(TLink[] sequence)
66     {
67         var indexed = true;
68         var i = sequence.Length;
69         var links = _links.Unsync;
70         while (--i >= 1 && (indexed =
71             ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i -
72             ↳ 1], sequence[i]), _null))) { }
73         for (; i >= 1; i--)
74         {
75             links.GetOrCreate(sequence[i - 1], sequence[i]);
76         }
77         return indexed;
78     }
79
80     public bool CheckIndex(IList<TLink> sequence)
81     {
82         var indexed = true;
83         var i = sequence.Count;
84         while (--i >= 1 && (indexed =
85             ↳ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i -
86             ↳ 1], sequence[i]), _null))) { }
87         return indexed;
88     }

```

```

81     }
82 }

./Sequences/SequencesOptions.cs
1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
5  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
6  using Platform.Data.Doublets.Sequences.Converters;
7  using Platform.Data.Doublets.Sequences.CreteriaMatchers;
8
9  namespace Platform.Data.Doublets.Sequences
10 {
11     public class SequencesOptions<TLink> // TODO: To use type parameter
12     ↪ <TLink> the ILinks<TLink> must contain GetConstants function.
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15         ↪ EqualityComparer<TLink>.Default;
16
17         public TLink SequenceMarkerLink { get; set; }
18         public bool UseCascadeUpdate { get; set; }
19         public bool UseCascadeDelete { get; set; }
20         public bool UseIndex { get; set; } // TODO: Update Index on sequence
21         ↪ update/delete.
22         public bool UseSequenceMarker { get; set; }
23         public bool UseCompression { get; set; }
24         public bool UseGarbageCollection { get; set; }
25         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get;
26         ↪ set; }
27         public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
28
29         public MarkedSequenceCreteriaMatcher<TLink> MarkedSequenceMatcher {
30         ↪ get; set; }
31         public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get;
32         ↪ set; }
33         public SequencesIndexer<TLink> Indexer { get; set; }
34
35         // TODO: Реализовать компактификацию при чтении
36         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
37         //public bool UseRequestMarker { get; set; }
38         //public bool StoreRequestResults { get; set; }
39
40         public void InitOptions(ISynchronizedLinks<TLink> links)
41         {
42             if (UseSequenceMarker)
43             {
44                 if (_equalityComparer.Equals(SequenceMarkerLink,
45                 ↪ links.Constants.Null))
46                 {
47                     SequenceMarkerLink = links.CreatePoint();
48                 }
49                 else
50                 {
51                     if (!links.Exists(SequenceMarkerLink))
52                     {
53                         var link = links.CreatePoint();
54                         if (!_equalityComparer.Equals(link,
55                         ↪ SequenceMarkerLink))
56                         {
57                             throw new InvalidOperationException("Cannot
58                             ↪ recreate sequence marker link.");
59                         }
60                     }
61                 }
62             }
63             if (MarkedSequenceMatcher == null)

```

```

54 {
55     MarkedSequenceMatcher = new
56     ↪ MarkedSequenceCreteriaMatcher<TLink>(links,
57     ↪ SequenceMarkerLink);
58 }
59 }
60 var balancedVariantConverter = new
61 ↪ BalancedVariantConverter<TLink>(links);
62 if (UseCompression)
63 {
64     if (LinksToSequenceConverter == null)
65     {
66         ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
67         if (UseSequenceMarker)
68         {
69             totalSequenceSymbolFrequencyCounter = new TotalMarkedS
70             ↪ equenceSymbolFrequencyCounter<TLink>(links,
71             ↪ MarkedSequenceMatcher);
72         }
73         else
74         {
75             totalSequenceSymbolFrequencyCounter = new
76             ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
77         }
78         var doubletFrequenciesCache = new
79         ↪ LinkFrequenciesCache<TLink>(links,
80         ↪ totalSequenceSymbolFrequencyCounter);
81         var compressingConverter = new
82         ↪ CompressingConverter<TLink>(links,
83         ↪ balancedVariantConverter, doubletFrequenciesCache);
84         LinksToSequenceConverter = compressingConverter;
85     }
86 }
87 else
88 {
89     if (LinksToSequenceConverter == null)
90     {
91         LinksToSequenceConverter = balancedVariantConverter;
92     }
93 }
94 if (UseIndex && Indexer == null)
95 {
96     Indexer = new SequencesIndexer<TLink>(links);
97 }
98 }

```

./Sequences/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 namespace Platform.Data.Doublets.Sequences
9 {
10     public class UnicodeMap
11     {
12         public static readonly ulong FirstCharLink = 1;
13         public static readonly ulong LastCharLink = FirstCharLink +
14             ↪ char.MaxValue;
15         public static readonly ulong MapSize = 1 + char.MaxValue;
16
17         private readonly ILinks<ulong> _links;
18         private bool _initialized;
19
20         public UnicodeMap(ILinks<ulong> links) => _links = links;
21
22         public static UnicodeMap InitNew(ILinks<ulong> links)
23         {
24             var map = new UnicodeMap(links);
25             map.Init();
26             return map;
27         }
28
29         public void Init()
30         {
31             if (_initialized)
32             {
33                 return;
34             }
35             _initialized = true;
36             var firstLink = _links.CreatePoint();
37             if (firstLink != FirstCharLink)
38             {
39                 _links.Delete(firstLink);
40             }
41             else
42             {
43                 for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
44                 {
45                     // From NIL to It (NIL -> Character) transformation
46                     ↪ meaning, (or infinite amount of NIL characters before
47                     ↪ actual Character)
48                     var createdLink = _links.CreatePoint();
49                     _links.Update(createdLink, firstLink, createdLink);
50                     if (createdLink != i)
51                     {
52                         throw new InvalidOperationException("Unable to
53                             ↪ initialize UTF 16 table.");
54                     }
55                 }
56             }
57         }
58
59         // 0 - null link
60         // 1 - nil character (0 character)
61         // ...
62         // 65536 (0(1) + 65535 = 65536 possible values)
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         public static ulong FromCharToLink(char character) => (ulong)character
66             ↪ + 1;
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         public static char FromLinkToChar(ulong link) => (char)(link - 1);
70
71         [MethodImpl(MethodImplOptions.AggressiveInlining)]
72         public static bool IsCharLink(ulong link) => link <= MapSize;

```

```

69
70 public static string FromLinksToString(IList<ulong> linksList)
71 {
72     var sb = new StringBuilder();
73     for (int i = 0; i < linksList.Count; i++)
74     {
75         sb.Append(FromLinkToChar(linksList[i]));
76     }
77     return sb.ToString();
78 }
79
80 public static string FromSequenceLinkToString(ulong link,
81     ↪ ILinks<ulong> links)
82 {
83     var sb = new StringBuilder();
84     if (links.Exists(link))
85     {
86         StopableSequenceWalker.WalkRight(link, links.GetSource,
87             ↪ links.GetTarget,
88             x => x <= MapSize || links.GetSource(x) == x ||
89             ↪ links.GetTarget(x) == x, element =>
90             {
91                 sb.Append(FromLinkToChar(element));
92                 return true;
93             }
94         );
95     }
96     return sb.ToString();
97 }
98
99 public static ulong[] FromCharsToLinkArray(char[] chars) =>
100     ↪ FromCharsToLinkArray(chars, chars.Length);
101
102 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
103 {
104     // char array to ulong array
105     var linksSequence = new ulong[count];
106     for (var i = 0; i < count; i++)
107     {
108         linksSequence[i] = FromCharToLink(chars[i]);
109     }
110     return linksSequence;
111 }
112
113 public static ulong[] FromStringToLinkArray(string sequence)
114 {
115     // char array to ulong array
116     var linksSequence = new ulong[sequence.Length];
117     for (var i = 0; i < sequence.Length; i++)
118     {
119         linksSequence[i] = FromCharToLink(sequence[i]);
120     }
121     return linksSequence;
122 }
123
124 public static List<ulong[]> FromStringToLinkArrayGroups(string
125     ↪ sequence)
126 {
127     var result = new List<ulong[]>();
128     var offset = 0;
129     while (offset < sequence.Length)
130     {
131         var currentCategory =
132             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
133         var relativeLength = 1;
134         var absoluteLength = offset + relativeLength;
135         while (absoluteLength < sequence.Length &&

```

```

128         currentCategory == CharUnicodeInfo.GetUnicodeCategory(s
            ↪     equence[absoluteLength]))
129     {
130         relativeLength++;
131         absoluteLength++;
132     }
133     // char array to ulong array
134     var innerSequence = new ulong[relativeLength];
135     var maxLength = offset + relativeLength;
136     for (var i = offset; i < maxLength; i++)
137     {
138         innerSequence[i - offset] = FromCharToLink(sequence[i]);
139     }
140     result.Add(innerSequence);
141     offset += relativeLength;
142 }
143 return result;
144 }
145
146 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[]
    ↪     array)
147 {
148     var result = new List<ulong[]>();
149     var offset = 0;
150     while (offset < array.Length)
151     {
152         var relativeLength = 1;
153         if (array[offset] <= LastCharLink)
154         {
155             var currentCategory = CharUnicodeInfo.GetUnicodeCategory(F
            ↪     romLinkToChar(array[offset]));
156             var absoluteLength = offset + relativeLength;
157             while (absoluteLength < array.Length &&
158                 array[absoluteLength] <= LastCharLink &&
159                 currentCategory == CharUnicodeInfo.GetUnicodeCatego
            ↪     ry(FromLinkToChar(array[absoluteLength])))
160             {
161                 relativeLength++;
162                 absoluteLength++;
163             }
164         }
165         else
166         {
167             var absoluteLength = offset + relativeLength;
168             while (absoluteLength < array.Length &&
            ↪     array[absoluteLength] > LastCharLink)
169             {
170                 relativeLength++;
171                 absoluteLength++;
172             }
173         }
174         // copy array
175         var innerSequence = new ulong[relativeLength];
176         var maxLength = offset + relativeLength;
177         for (var i = offset; i < maxLength; i++)
178         {
179             innerSequence[i - offset] = array[i];
180         }
181         result.Add(innerSequence);
182         offset += relativeLength;
183     }
184     return result;
185 }
186 }
187 }

```

./Sequences/Walkers/LeftSequenceWalker.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 namespace Platform.Data.Doublets.Sequences.Walkers
5 {
6     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
7     {
8         public LeftSequenceWalker(ILinks<TLink> links) : base(links) { }
9
10        [MethodImpl(MethodImplOptions.AggressiveInlining)]
11        protected override IList<TLink> GetNextElementAfterPop(IList<TLink>
            ↪     element) => Links.GetLink(Links.GetSource(element));
12
13        [MethodImpl(MethodImplOptions.AggressiveInlining)]
14        protected override IList<TLink> GetNextElementAfterPush(IList<TLink>
            ↪     element) => Links.GetLink(Links.GetTarget(element));
15
16        [MethodImpl(MethodImplOptions.AggressiveInlining)]
17        protected override IEnumerable<IList<TLink>> WalkContents(IList<TLink>
            ↪     element)
18        {
19            var start = Links.Constants.IndexPart + 1;
20            for (var i = element.Count - 1; i >= start; i--)
21            {
22                var partLink = Links.GetLink(element[i]);
23                if (IsElement(partLink))
24                {
25                    yield return partLink;
26                }
27            }
28        }
29    }
30 }

```

./Sequences/Walkers/RightSequenceWalker.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 namespace Platform.Data.Doublets.Sequences.Walkers
5 {
6     public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
7     {
8         public RightSequenceWalker(ILinks<TLink> links) : base(links) { }
9
10        [MethodImpl(MethodImplOptions.AggressiveInlining)]
11        protected override IList<TLink> GetNextElementAfterPop(IList<TLink>
            ↪     element) => Links.GetLink(Links.GetTarget(element));
12
13        [MethodImpl(MethodImplOptions.AggressiveInlining)]
14        protected override IList<TLink> GetNextElementAfterPush(IList<TLink>
            ↪     element) => Links.GetLink(Links.GetSource(element));
15
16        [MethodImpl(MethodImplOptions.AggressiveInlining)]
17        protected override IEnumerable<IList<TLink>> WalkContents(IList<TLink>
            ↪     element)
18        {
19            for (var i = Links.Constants.IndexPart + 1; i < element.Count; i++)
20            {
21                var partLink = Links.GetLink(element[i]);
22                if (IsElement(partLink))
23                {
24                    yield return partLink;
25                }
26            }
27        }
28    }
29 }

```

```

27     }
28 }
29 }

./Sequences/Walkers/SequenceWalkerBase.cs
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Data.Sequences;
4
5 namespace Platform.Data.Doublets.Sequences.Walkers
6 {
7     public abstract class SequenceWalkerBase<TLink> :
8         ↳ LinksOperatorBase<TLink>, ISequenceWalker<TLink>
9     {
10         // TODO: Use IStack instead of System.Collections.Generic.Stack, but
11         ↳ IStack should contain IsEmpty property
12         private readonly Stack<IList<TLink>> _stack;
13
14         protected SequenceWalkerBase(ILinks<TLink> links) : base(links) =>
15             ↳ _stack = new Stack<IList<TLink>>();
16
17         public IEnumerable<IList<TLink>> Walk(TLink sequence)
18         {
19             if (_stack.Count > 0)
20             {
21                 _stack.Clear(); // This can be replaced with
22                 ↳ while(!_stack.IsEmpty) _stack.Pop()
23             }
24             var element = Links.GetLink(sequence);
25             if (IsElement(element))
26             {
27                 yield return element;
28             }
29             else
30             {
31                 while (true)
32                 {
33                     if (IsElement(element))
34                     {
35                         if (_stack.Count == 0)
36                         {
37                             break;
38                         }
39                         element = _stack.Pop();
40                         foreach (var output in WalkContents(element))
41                         {
42                             yield return output;
43                         }
44                         element = GetNextElementAfterPop(element);
45                     }
46                     else
47                     {
48                         _stack.Push(element);
49                         element = GetNextElementAfterPush(element);
50                     }
51                 }
52             }
53
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             protected virtual bool IsElement(IList<TLink> elementLink) =>
56                 ↳ Point<TLink>.IsPartialPointUnchecked(elementLink);
57
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]
59             protected abstract IList<TLink> GetNextElementAfterPop(IList<TLink>
60                 ↳ element);

```

```

56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected abstract IList<TLink> GetNextElementAfterPush(IList<TLink>
59             ↳ element);
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected abstract IEnumerable<IList<TLink>> WalkContents(IList<TLink>
63             ↳ element);
64     }
65 }

```

```

./Stacks/Stack.cs
1 using System.Collections.Generic;
2 using Platform.Collections.Stacks;
3
4 namespace Platform.Data.Doublets.Stacks
5 {
6     public class Stack<TLink> : IStack<TLink>
7     {
8         private static readonly EqualityComparer<TLink> _equalityComparer =
9             ↳ EqualityComparer<TLink>.Default;
10
11         private readonly ILinks<TLink> _links;
12         private readonly TLink _stack;
13
14         public Stack(ILinks<TLink> links, TLink stack)
15         {
16             _links = links;
17             _stack = stack;
18         }
19
20         private TLink GetStackMarker() => _links.GetSource(_stack);
21
22         private TLink GetTop() => _links.GetTarget(_stack);
23
24         public TLink Peek() => _links.GetTarget(GetTop());
25
26         public TLink Pop()
27         {
28             var element = Peek();
29             if (!_equalityComparer.Equals(element, _stack))
30             {
31                 var top = GetTop();
32                 var previousTop = _links.GetSource(top);
33                 _links.Update(_stack, GetStackMarker(), previousTop);
34                 _links.Delete(top);
35             }
36             return element;
37         }
38
39         public void Push(TLink element) => _links.Update(_stack,
40             ↳ GetStackMarker(), _links.GetOrCreate(GetTop(), element));
41     }
42 }

```

```

./Stacks/StackExtensions.cs
1 namespace Platform.Data.Doublets.Stacks
2 {
3     public static class StackExtensions
4     {
5         public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink
6             ↳ stackMarker)
7         {
8             var stackPoint = links.CreatePoint();
9             var stack = links.Update(stackPoint, stackMarker, stackPoint);
10         }
11     }
12 }

```

```

9         return stack;
10     }
11
12     public static void DeleteStack<TLink>(this ILinks<TLink> links, TLink
13     ↪ stack) => links.Delete(stack);
14 }

```

./SynchronizedLinks.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Data.Constants;
4 using Platform.Data.Doublets;
5 using Platform.Threading.Synchronization;
6
7 namespace Platform.Data.Doublets
8 {
9     /// <remarks>
10    /// TODO: Autogeneration of synchronized wrapper (decorator).
11    /// TODO: Try to unfold code of each method using IL generation for
12    ↪ performance improvements.
13    /// TODO: Or even to unfold multiple layers of implementations.
14    /// </remarks>
15    public class SynchronizedLinks<T> : ISynchronizedLinks<T>
16    {
17        public LinksCombinedConstants<T, T, int> Constants { get; }
18        public ISynchronization SyncRoot { get; }
19        public ILinks<T> Sync { get; }
20        public ILinks<T> Unsync { get; }
21
22        public SynchronizedLinks(ILinks<T> links) : this(new
23        ↪ ReaderWriterLockSynchronization(), links) { }
24
25        public SynchronizedLinks(ISynchronization synchronization, ILinks<T>
26        ↪ links)
27        {
28            SyncRoot = synchronization;
29            Sync = this;
30            Unsync = links;
31            Constants = links.Constants;
32
33            public T Count(IList<T> restriction) =>
34            ↪ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
35            public T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
36            ↪ SyncRoot.ExecuteReadOperation(handler, restrictions, (handler1,
37            ↪ restrictions1) => Unsync.Each(handler1, restrictions1));
38            public T Create() => SyncRoot.ExecuteWriteOperation(Unsync.Create);
39            public T Update(IList<T> restrictions) =>
40            ↪ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Update);
41            public void Delete(T link) => SyncRoot.ExecuteWriteOperation(link,
42            ↪ Unsync.Delete);
43
44            //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T>
45            ↪ matchedHandler, IList<T> substitution, Func<IList<T>, IList<T>, T>
46            ↪ substitutedHandler)
47            //{
48            //    if (restriction != null && substitution != null &&
49            ↪ !substitution.EqualTo(restriction))
50            //        return SyncRoot.ExecuteWriteOperation(restriction,
51            ↪ matchedHandler, substitution, substitutedHandler, Unsync.Trigger);
52            //    return SyncRoot.ExecuteReadOperation(restriction,
53            ↪ matchedHandler, substitution, substitutedHandler, Unsync.Trigger);
54            //}

```

```

44     }
45 }

```

./UInt64Link.cs

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using Platform.Exceptions;
5 using Platform.Ranges;
6 using Platform.Helpers.Singletons;
7 using Platform.Data.Constants;
8
9 namespace Platform.Data.Doublets
10 {
11     /// <summary>
12     /// Структура описывающая уникальную связь.
13     /// </summary>
14     public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>,
15     ↪ IList<ulong>
16     {
17         private static readonly LinksCombinedConstants<bool, ulong, int>
18         ↪ _constants = Default<LinksCombinedConstants<bool, ulong,
19         ↪ int>>.Instance;
20
21         private const int Length = 3;
22
23         public readonly ulong Index;
24         public readonly ulong Source;
25         public readonly ulong Target;
26
27         public static readonly UInt64Link Null = new UInt64Link();
28
29         public UInt64Link(params ulong[] values)
30         {
31             Index = values.Length > _constants.IndexPart ?
32             ↪ values[_constants.IndexPart] : _constants.Null;
33             Source = values.Length > _constants.SourcePart ?
34             ↪ values[_constants.SourcePart] : _constants.Null;
35             Target = values.Length > _constants.TargetPart ?
36             ↪ values[_constants.TargetPart] : _constants.Null;
37         }
38
39         public UInt64Link(IList<ulong> values)
40         {
41             Index = values.Count > _constants.IndexPart ?
42             ↪ values[_constants.IndexPart] : _constants.Null;
43             Source = values.Count > _constants.SourcePart ?
44             ↪ values[_constants.SourcePart] : _constants.Null;
45             Target = values.Count > _constants.TargetPart ?
46             ↪ values[_constants.TargetPart] : _constants.Null;
47         }
48
49         public UInt64Link(ulong index, ulong source, ulong target)
50         {
51             Index = index;
52             Source = source;
53             Target = target;
54         }
55
56         public UInt64Link(ulong source, ulong target)
57         : this(_constants.Null, source, target)
58         {
59             Source = source;
60             Target = target;
61         }
62
63         public static UInt64Link Create(ulong source, ulong target) => new
64         ↪ UInt64Link(source, target);

```

```

55     public override int GetHashCode() => (Index, Source,
56         ↪ Target).GetHashCode();

57
58     public bool IsNull() => Index == _constants.Null
59         && Source == _constants.Null
60         && Target == _constants.Null;
61
62     public override bool Equals(object other) => other is UInt64Link &&
63         ↪ Equals((UInt64Link)other);
64
65     public bool Equals(UInt64Link other) => Index == other.Index
66         && Source == other.Source
67         && Target == other.Target;
68
69     public static string ToString(ulong index, ulong source, ulong target)
70     ↪ => $"({index}: {source}->{target})";
71
72     public static string ToString(ulong source, ulong target) =>
73     ↪ $"({source}->{target})";
74
75     public static implicit operator ulong[](UInt64Link link) =>
76     ↪ link.ToArray();
77
78     public static implicit operator UInt64Link(ulong[] linkArray) => new
79     ↪ UInt64Link(linkArray);
80
81     public ulong[] ToArray()
82     {
83         var array = new ulong[Length];
84         CopyTo(array, 0);
85         return array;
86     }
87
88     public override string ToString() => Index == _constants.Null ?
89     ↪ ToString(Source, Target) : ToString(Index, Source, Target);
90
91     #region IList
92
93     public ulong this[int index]
94     {
95         get
96         {
97             Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length
98                 ↪ - 1), nameof(index));
99             if (index == _constants.IndexPart)
100             {
101                 return Index;
102             }
103             if (index == _constants.SourcePart)
104             {
105                 return Source;
106             }
107             if (index == _constants.TargetPart)
108             {
109                 return Target;
110             }
111             throw new NotSupportedException(); // Impossible path due to
112                 ↪ Ensure.ArgumentInRange
113         }
114         set => throw new NotSupportedException();
115     }
116
117     public int Count => Length;
118
119     public bool IsReadOnly => true;

```

```

112
113     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
114
115     public IEnumerator<ulong> GetEnumerator()
116     {
117         yield return Index;
118         yield return Source;
119         yield return Target;
120     }
121
122     public void Add(ulong item) => throw new NotSupportedException();
123
124     public void Clear() => throw new NotSupportedException();
125
126     public bool Contains(ulong item) => IndexOf(item) >= 0;
127
128     public void CopyTo(ulong[] array, int arrayIndex)
129     {
130         Ensure.Always.ArgumentNotNull(array, nameof(array));
131         Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0,
132             ↪ array.Length - 1), nameof(arrayIndex));
133         if (arrayIndex + Length > array.Length)
134         {
135             throw new ArgumentException();
136         }
137         array[arrayIndex++] = Index;
138         array[arrayIndex++] = Source;
139         array[arrayIndex] = Target;
140     }
141
142     public bool Remove(ulong item) =>
143     ↪ Throw.A.NotSupportedExceptionAndReturn<bool>();
144
145     public int IndexOf(ulong item)
146     {
147         if (Index == item)
148         {
149             return _constants.IndexPart;
150         }
151         if (Source == item)
152         {
153             return _constants.SourcePart;
154         }
155         if (Target == item)
156         {
157             return _constants.TargetPart;
158         }
159         return -1;
160     }
161
162     public void Insert(int index, ulong item) => throw new
163     ↪ NotSupportedException();
164
165     public void RemoveAt(int index) => throw new NotSupportedException();
166
167     #endregion
168 }

```

./UInt64LinkExtensions.cs

```

1 namespace Platform.Data.Doublets
2 {
3     public static class UInt64LinkExtensions
4     {
5         public static bool IsFullPoint(this UInt64Link link) =>
6             ↪ Point<ulong>.IsFullPoint(link);

```



```

6         public static bool IsPartialPoint(this UInt64Link link) =>
7             ↳ Point<ulong>.IsPartialPoint(link);
8     }
}

./UInt64LinksExtensions.cs
1 using System;
2 using System.Text;
3 using System.Collections.Generic;
4 using Platform.Helpers.Singletons;
5 using Platform.Data.Constants;
6 using Platform.Data.Exceptions;
7 using Platform.Data.Doublets.Sequences;
8
9 namespace Platform.Data.Doublets
10 {
11     public static class UInt64LinksExtensions
12     {
13         public static readonly LinksCombinedConstants<bool, ulong, int>
14             ↳ Constants = Default<LinksCombinedConstants<bool, ulong,
15             ↳ int>>.Instance;
16
17         public static void UseUnicode(this ILinks<ulong> links) =>
18             ↳ UnicodeMap.InitNew(links);
19
20         public static void EnsureEachLinkExists(this ILinks<ulong> links,
21             ↳ IList<ulong> sequence)
22         {
23             if (sequence == null)
24             {
25                 return;
26             }
27             for (var i = 0; i < sequence.Count; i++)
28             {
29                 if (!links.Exists(sequence[i]))
30                 {
31                     throw new
32                         ↳ ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
33                         ↳ $"sequence[{i}]");
34                 }
35             }
36         }
37
38         public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong>
39             ↳ links, IList<ulong> sequence)
40         {
41             if (sequence == null)
42             {
43                 return;
44             }
45             for (var i = 0; i < sequence.Count; i++)
46             {
47                 if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
48                 {
49                     throw new
50                         ↳ ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
51                         ↳ $"sequence[{i}]");
52                 }
53             }
54         }
55
56         public static bool AnyLinkIsAny(this ILinks<ulong> links, params
57             ↳ ulong[] sequence)
58         {
59             if (sequence == null)
60             {

```

```

51         return false;
52     }
53     var constants = links.Constants;
54     for (var i = 0; i < sequence.Length; i++)
55     {
56         if (sequence[i] == constants.Any)
57         {
58             return true;
59         }
60     }
61     return false;
62 }
63
64 public static string FormatStructure(this ILinks<ulong> links, ulong
65     ↳ linkIndex, Func<UInt64Link, bool> isElement, bool renderIndex =
66     ↳ false, bool renderDebug = false)
67 {
68     var sb = new StringBuilder();
69     var visited = new HashSet<ulong>();
70     links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb,
71     ↳ link) => innerSb.Append(link.Index), renderIndex, renderDebug);
72     return sb.ToString();
73 }
74
75 public static string FormatStructure(this ILinks<ulong> links, ulong
76     ↳ linkIndex, Func<UInt64Link, bool> isElement, Action<StringBuilder,
77     ↳ UInt64Link> appendElement, bool renderIndex = false, bool
78     ↳ renderDebug = false)
79 {
80     var sb = new StringBuilder();
81     var visited = new HashSet<ulong>();
82     links.AppendStructure(sb, visited, linkIndex, isElement,
83     ↳ appendElement, renderIndex, renderDebug);
84     return sb.ToString();
85 }
86
87 public static void AppendStructure(this ILinks<ulong> links,
88     ↳ StringBuilder sb, HashSet<ulong> visited, ulong linkIndex,
89     ↳ Func<UInt64Link, bool> isElement, Action<StringBuilder,
90     ↳ UInt64Link> appendElement, bool renderIndex = false, bool
91     ↳ renderDebug = false)
92 {
93     if (sb == null)
94     {
95         throw new ArgumentNullException(nameof(sb));
96     }
97     if (linkIndex == Constants.Null || linkIndex == Constants.Any ||
98     ↳ linkIndex == Constants.Itself)
99     {
100         return;
101     }
102     if (links.Exists(linkIndex))
103     {
104         if (visited.Add(linkIndex))
105         {
106             sb.Append('(');
107             var link = new UInt64Link(links.GetLink(linkIndex));
108             if (renderIndex)
109             {
110                 sb.Append(link.Index);
111                 sb.Append(':');
112             }
113             if (link.Source == link.Index)
114             {
115                 sb.Append(link.Index);

```

```

104     }
105     else
106     {
107         var source = new
            ↳ UInt64Link(links.GetLink(link.Source));
108         if (isElement(source))
109         {
110             appendElement(sb, source);
111         }
112         else
113         {
114             links.AppendStructure(sb, visited, source.Index,
            ↳ isElement, appendElement, renderIndex);
115         }
116     }
117     sb.Append(' ');
118     if (link.Target == link.Index)
119     {
120         sb.Append(link.Index);
121     }
122     else
123     {
124         var target = new
            ↳ UInt64Link(links.GetLink(link.Target));
125         if (isElement(target))
126         {
127             appendElement(sb, target);
128         }
129         else
130         {
131             links.AppendStructure(sb, visited, target.Index,
            ↳ isElement, appendElement, renderIndex);
132         }
133     }
134     sb.Append(' ');
135 }
136 else
137 {
138     if (renderDebug)
139     {
140         sb.Append('*');
141     }
142     sb.Append(linkIndex);
143 }
144 }
145 else
146 {
147     if (renderDebug)
148     {
149         sb.Append('~');
150     }
151     sb.Append(linkIndex);
152 }
153 }
154 }
155 }

```

./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
14 namespace Platform.Data.Doublets
15 {
16     public class UInt64LinksTransactionsLayer :
            ↳ LinksDisposableDecoratorBase<ulong> //-V3073
17     {
18         /// <remarks>
19         /// Альтернативные варианты хранения трансформации (элемента
            ↳ транзакции):
20         ///
21         /// private enum TransitionType
22         /// {
23         ///     Creation,
24         ///     UpdateOf,
25         ///     UpdateTo,
26         ///     Deletion
27         /// }
28         ///
29         /// private struct Transition
30         /// {
31         ///     public ulong TransactionId;
32         ///     public UniqueTimestamp Timestamp;
33         ///     public TransactionItemType Type;
34         ///     public Link Source;
35         ///     public Link Linker;
36         ///     public Link Target;
37         /// }
38         ///
39         /// Или
40         ///
41         /// public struct TransitionHeader
42         /// {
43         ///     public ulong TransactionIdCombined;
44         ///     public ulong TimestampCombined;
45         ///
46         ///     public ulong TransactionId
47         ///     {
48         ///         get
49         ///         {
50             return (ulong) mask & TransactionIdCombined;
51         }
52         }
53         ///
54         ///     public UniqueTimestamp Timestamp
55         ///     {
56         ///         get
57         ///         {
58             return (UniqueTimestamp)mask & TransactionIdCombined;
59         }
60         }
61         ///
62         ///     public TransactionItemType Type
63         ///     {
64         ///         get
65         ///         {
66             // Использовать по одному биту из TransactionId и
            ↳ Timestamp,
67             // для значения в 2 бита, которое представляет тип
            ↳ операции
68             throw new NotImplementedException();
69         }

```

```

70     /// }
71     /// }
72     ///
73     /// private struct Transition
74     /// {
75     ///     public TransitionHeader Header;
76     ///     public Link Source;
77     ///     public Link Linker;
78     ///     public Link Target;
79     /// }
80     ///
81     /// </remarks>
82     public struct Transition
83     {
84         public static readonly long Size =
85             ↳ StructureHelpers.SizeOf<Transition>();
86
87         public readonly ulong TransactionId;
88         public readonly UInt64Link Before;
89         public readonly UInt64Link After;
90         public readonly Timestamp Timestamp;
91
92         public Transition(UniqueTimestampFactory uniqueTimestampFactory,
93             ↳ ulong transactionId, UInt64Link before, UInt64Link after)
94         {
95             TransactionId = transactionId;
96             Before = before;
97             After = after;
98             Timestamp = uniqueTimestampFactory.Create();
99         }
100
101         public Transition(UniqueTimestampFactory uniqueTimestampFactory,
102             ↳ ulong transactionId, UInt64Link before)
103         : this(uniqueTimestampFactory, transactionId, before, default)
104         {
105         }
106
107         public Transition(UniqueTimestampFactory uniqueTimestampFactory,
108             ↳ ulong transactionId)
109         : this(uniqueTimestampFactory, transactionId, default, default)
110         {
111         }
112
113         public override string ToString() => $"{Timestamp}
114             ↳ {TransactionId}: {Before} => {After}";
115     }
116
117     /// <remarks>
118     /// Другие варианты реализации транзакций (атомарности):
119     /// 1. Разделение хранения значения связи ((Source Target) или
120     ///     ↳ (Source Linker Target)) и индексов.
121     /// 2. Хранение трансформаций/операций в отдельном хранилище
122     ///     ↳ Links, но дополнительно потребуется решить вопрос
123     ///     ↳ со ссылками на внешние идентификаторы, или как-то иначе
124     ///     ↳ решить вопрос с пересечениями идентификаторов.
125     ///
126     /// Где хранить промежуточный список транзакций?
127     ///
128     /// В оперативной памяти:
129     /// Минусы:
130     /// 1. Может усложнить систему, если она будет функционировать
131     ///     ↳ самостоятельно,
132     /// так как нужно отдельно выделять память под список
133     ↳ трансформаций.
134     /// 2. Выделенной оперативной памяти может не хватить, в том
135     ↳ случае,

```

```

125     ///     если транзакция использует слишком много трансформаций.
126     ///     -> Можно использовать жёсткий диск для слишком длинных
127     ↳ транзакций.
128     ///     -> Максимальный размер списка трансформаций можно
129     ↳ ограничить / задать константой.
130     /// 3. При подтверждении транзакции (Commit) все трансформации
131     ↳ записываются разом создавая задержку.
132     ///
133     /// На жёстком диске:
134     /// Минусы:
135     /// 1. Длительный отклик, на запись каждой трансформации.
136     /// 2. Лог транзакций дополнительно наполняется отменёнными
137     ↳ транзакциями.
138     ///     -> Это может решаться упаковкой/исключением дублирующих
139     ↳ операций.
140     ///     -> Также это может решаться тем, что короткие транзакции
141     ↳ вообще
142     ///     не будут записываться в случае отката.
143     /// 3. Перед тем как выполнять отмену операций транзакции нужно
144     ↳ дожидаться пока все операции (трансформации)
145     ↳ будут записаны в лог.
146     /// </remarks>
147     public class Transaction : DisposableBase
148     {
149         private readonly Queue<Transition> _transitions;
150         private readonly UInt64LinksTransactionsLayer _layer;
151         public bool IsCommitted { get; private set; }
152         public bool IsReverted { get; private set; }
153
154         public Transaction(UInt64LinksTransactionsLayer layer)
155         {
156             _layer = layer;
157             if (_layer._currentTransactionId != 0)
158             {
159                 throw new NotSupportedException("Nested transactions not
160                     ↳ supported.");
161             }
162             IsCommitted = false;
163             IsReverted = false;
164             _transitions = new Queue<Transition>();
165             SetCurrentTransaction(layer, this);
166         }
167
168         public void Commit()
169         {
170             EnsureTransactionAllowsWriteOperations(this);
171             while (_transitions.Count > 0)
172             {
173                 var transition = _transitions.Dequeue();
174                 _layer._transitions.Enqueue(transition);
175             }
176             _layer._lastCommittedTransactionId =
177                 ↳ _layer._currentTransactionId;
178             IsCommitted = true;
179         }
180
181         private void Revert()
182         {
183             EnsureTransactionAllowsWriteOperations(this);
184             var transitionsToRevert = new Transition[_transitions.Count];
185             _transitions.CopyTo(transitionsToRevert, 0);
186             for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
187             {
188                 _layer.RevertTransition(transitionsToRevert[i]);
189             }
190         }

```

```

181     }
182     IsReverted = true;
183 }
184
185 public static void
186     ↳ SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
187     ↳ Transaction transaction)
188 {
189     layer._currentTransactionId = layer._lastCommittedTransactionId
190     ↳ + 1;
191     layer._currentTransactionTransitions =
192     ↳ transaction._transitions;
193     layer._currentTransaction = transaction;
194 }
195
196 public static void
197     ↳ EnsureTransactionAllowsWriteOperations(Transaction transaction)
198 {
199     if (transaction.IsReverted)
200     {
201         throw new InvalidOperationException("Transation is
202         ↳ reverted.");
203     }
204     if (transaction.IsCommitted)
205     {
206         throw new InvalidOperationException("Transation is
207         ↳ committed.");
208     }
209 }
210
211 protected override void DisposeCore(bool manual, bool wasDisposed)
212 {
213     if (!wasDisposed && _layer != null && !_layer.IsDisposed)
214     {
215         if (!IsCommitted && !IsReverted)
216         {
217             Revert();
218         }
219         _layer.ResetCurrentTransation();
220     }
221 }
222
223 // TODO: THIS IS EXCEPTION WORKAROUND, REMOVE IT THEN
224 ↳ https://github.com/linksplatform/Disposables/issues/13 FIXED
225 protected override bool AllowMultipleDisposeCalls => true;
226 }
227
228 public static readonly TimeSpan DefaultPushDelay =
229     ↳ TimeSpan.FromSeconds(0.1);
230
231 private readonly string _logAddress;
232 private readonly FileStream _log;
233 private readonly Queue<Transition> _transitions;
234 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
235 private Task _transitionsPusher;
236 private Transition _lastCommittedTransition;
237 private ulong _currentTransactionId;
238 private Queue<Transition> _currentTransactionTransitions;
239 private Transaction _currentTransaction;
240 private ulong _lastCommittedTransactionId;
241
242 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string
243     ↳ logAddress)
244     : base(links)
245 {
246     if (string.IsNullOrEmpty(logAddress))

```

```

237 {
238     throw new ArgumentNullException(nameof(logAddress));
239 }
240
241 // В первой строке файла хранится последняя закоммиченную
242 ↳ транзакцию.
243 // При запуске это используется для проверки удачного закрытия
244 ↳ файла лора.
245 // In the first line of the file the last committed transaction is
246 ↳ stored.
247 // On startup, this is used to check that the log file is
248 ↳ successfully closed.
249 var lastCommittedTransition =
250     ↳ FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
251 var lastWrittenTransition =
252     ↳ FileHelpers.ReadLastOrDefault<Transition>(logAddress);
253 if (!lastCommittedTransition.Equals(lastWrittenTransition))
254 {
255     Dispose();
256     throw new NotSupportedException("Database is damaged,
257     ↳ autorecovery is not supported yet.");
258 }
259 if (lastCommittedTransition.Equals(default(Transition)))
260 {
261     FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
262 }
263 _lastCommittedTransition = lastCommittedTransition;
264 // TODO: Think about a better way to calculate or store this value
265 var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
266 _lastCommittedTransactionId = allTransitions.Max(x =>
267     ↳ x.TransactionId);
268 _uniqueTimestampFactory = new UniqueTimestampFactory();
269 _logAddress = logAddress;
270 _log = FileHelpers.Append(logAddress);
271 _transitions = new Queue<Transition>();
272 _transitionsPusher = new Task(TransitionsPusher);
273 _transitionsPusher.Start();
274 }
275
276 public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
277
278 public override ulong Create()
279 {
280     var createdLinkIndex = Links.Create();
281     var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
282     CommitTransition(new Transition(_uniqueTimestampFactory,
283     ↳ _currentTransactionId, default, createdLink));
284     return createdLinkIndex;
285 }
286
287 public override ulong Update(IList<ulong> parts)
288 {
289     var beforeLink = new
290     ↳ UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
291     parts[Constants.IndexPart] = Links.Update(parts);
292     var afterLink = new
293     ↳ UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
294     CommitTransition(new Transition(_uniqueTimestampFactory,
295     ↳ _currentTransactionId, beforeLink, afterLink));
296     return parts[Constants.IndexPart];
297 }
298
299 public override void Delete(ulong link)
300 {
301     var deletedLink = new UInt64Link(Links.GetLink(link));
302     Links.Delete(link);

```

```

290         CommitTransition(new Transition(_uniqueTimestampFactory,
291             ↳ _currentTransactionId, deletedLink, default));
292     }
293     [MethodImpl(MethodImplOptions.AggressiveInlining)]
294     private Queue<Transition> GetCurrentTransitions() =>
295         ↳ _currentTransactionTransitions ?? _transitions;
296     private void CommitTransition(Transition transition)
297     {
298         if (_currentTransaction != null)
299         {
300             Transaction.EnsureTransactionAllowsWriteOperations(_currentTra
301                 ↳ nsaction);
302         }
303         var transitions = GetCurrentTransitions();
304         transitions.Enqueue(transition);
305     }
306     private void RevertTransition(Transition transition)
307     {
308         if (transition.After.IsNull()) // Revert Deletion with Creation
309         {
310             Links.Create();
311         }
312         else if (transition.Before.IsNull()) // Revert Creation with
313             ↳ Deletion
314         {
315             Links.Delete(transition.After.Index);
316         }
317         else // Revert Update
318         {
319             Links.Update(new[] { transition.After.Index,
320                 ↳ transition.Before.Source, transition.Before.Target });
321         }
322     }
323     private void ResetCurrentTransation()
324     {
325         _currentTransactionId = 0;
326         _currentTransactionTransitions = null;
327         _currentTransaction = null;
328     }
329     private void PushTransitions()
330     {
331         if (_log == null || _transitions == null)
332         {
333             return;
334         }
335         for (var i = 0; i < _transitions.Count; i++)
336         {
337             var transition = _transitions.Dequeue();

```

```

338
339         _log.Write(transition);
340         _lastCommittedTransition = transition;
341     }
342     }
343     private void TransitionsPusher()
344     {
345         while (!IsDisposed && _transitionsPusher != null)
346         {
347             Thread.Sleep(DefaultPushDelay);
348             PushTransitions();
349         }
350     }
351     public Transaction BeginTransaction() => new Transaction(this);
352     private void DisposeTransitions()
353     {
354         try
355         {
356             var pusher = _transitionsPusher;
357             if (pusher != null)
358             {
359                 _transitionsPusher = null;
360                 pusher.Wait();
361             }
362             if (_transitions != null)
363             {
364                 PushTransitions();
365             }
366             Disposable.TryDispose(_log);
367             FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
368         }
369         catch
370         {
371         }
372     }
373     }
374     #region DisposalBase
375     protected override void DisposeCore(bool manual, bool wasDisposed)
376     {
377         if (!wasDisposed)
378         {
379             DisposeTransitions();
380         }
381         base.DisposeCore(manual, wasDisposed);
382     }
383     #endregion
384     }
385     }
386     }
387     }
388     }
389     }
390     }

```

Index

./Converters/AddressToUnaryNumberConverter.cs, 1
./Converters/LinkToltsFrequencyNumberConveter.cs, 1
./Converters/PowerOf2ToUnaryNumberConverter.cs, 1
./Converters/UnaryNumberToAddressAddOperationConverter.cs, 2
./Converters/UnaryNumberToAddressOrOperationConverter.cs, 2
./Decorators/LinksCascadeDependenciesResolver.cs, 3
./Decorators/LinksCascadeUniquenessAndDependenciesResolver.cs, 3
./Decorators/LinksDecoratorBase.cs, 3
./Decorators/LinksDependenciesValidator.cs, 4
./Decorators/LinksDisposableDecoratorBase.cs, 4
./Decorators/LinksInnerReferenceValidator.cs, 4
./Decorators/LinksNonExistentReferencesCreator.cs, 4
./Decorators/LinksNullToSelfReferenceResolver.cs, 5
./Decorators/LinksSelfReferenceResolver.cs, 5
./Decorators/LinksUniquenessResolver.cs, 5
./Decorators/LinksUniquenessValidator.cs, 6
./Decorators/NonNullContentsLinkDeletionResolver.cs, 6
./Decorators/UInt64Links.cs, 6
./Decorators/UniLinks.cs, 7
./Doublet.cs, 10
./DoubletComparer.cs, 10
./Hybrid.cs, 10
./ILinks.cs, 11
./ILinksExtensions.cs, 11
./ISynchronizedLinks.cs, 18
./Incrementers/FrequencyIncrementer.cs, 17
./Incrementers/LinkFrequencyIncrementer.cs, 18
./Incrementers/UnaryNumberIncrementer.cs, 18
./Link.cs, 18
./LinkExtensions.cs, 20
./LinksOperatorBase.cs, 20
./PropertyOperators/DefaultLinkPropertyOperator.cs, 20
./PropertyOperators/FrequencyPropertyOperator.cs, 20
./ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs, 27
./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs, 27
./ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 21
./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs, 36
./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs, 36
./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 31
./Sequences/Converters/BalancedVariantConverter.cs, 40
./Sequences/Converters/CompressingConverter.cs, 41
./Sequences/Converters/LinksListToSequenceConverterBase.cs, 43
./Sequences/Converters/OptimalVariantConverter.cs, 43
./Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 44
./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs, 44
./Sequences/CreteriaMatchers/MarkedSequenceCreteriaMatcher.cs, 44
./Sequences/DefaultSequenceAppender.cs, 44
./Sequences/DuplicateSegmentsCounter.cs, 45
./Sequences/DuplicateSegmentsProvider.cs, 45
./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs, 46
./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToltsFrequencyNumberConverter.cs, 47
./Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 47
./Sequences/Frequencies/Cache/LinkFrequency.cs, 48
./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 48
./Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 48
./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 49
./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 49
./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 49
./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 49
./Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 50
./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 50
./Sequences/HeightProviders/ISequenceHeightProvider.cs, 50
./Sequences/Sequences.Experiments.ReadSequence.cs, 73
./Sequences/Sequences.Experiments.cs, 57
./Sequences/Sequences.cs, 50
./Sequences/SequencesExtensions.cs, 74
./Sequences/SequencesIndexer.cs, 74
./Sequences/SequencesOptions.cs, 75
./Sequences/UnicodeMap.cs, 75
./Sequences/Walkers/LeftSequenceWalker.cs, 77
./Sequences/Walkers/RightSequenceWalker.cs, 77
./Sequences/Walkers/SequenceWalkerBase.cs, 78
./Stacks/Stack.cs, 78
./Stacks/StackExtensions.cs, 78
./SynchronizedLinks.cs, 79
./UInt64Link.cs, 79
./UInt64LinkExtensions.cs, 80
./UInt64LinksExtensions.cs, 81
./UInt64LinksTransactionsLayer.cs, 82
./obj/Debug/netstandard2.0/Platform.Data.Doublets.AssemblyInfo.cs, 20