

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> : LinksOperatorBase<TLink>, IConverter<TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ⇨ EqualityComparer<TLink>.Default;
12
13         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
14
15         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
16             ⇨ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
17             ⇨ powerOf2ToUnaryNumberConverter;
18
19         public TLink Convert(TLink sourceAddress)
20         {
21             var number = sourceAddress;
22             var target = Links.Constants.Null;
23             for (int i = 0; i < CachedTypeInfo<TLink>.BitsLength; i++)
24             {
25                 if (_equalityComparer.Equals(ArithmeticHelpers.And(number, Integer<TLink>.One),
26                     ⇨ Integer<TLink>.One))
27                 {
28                     target = _equalityComparer.Equals(target, Links.Constants.Null)
29                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
30                         : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
31                 }
32                 number = (Integer<TLink>)((ulong)(Integer<TLink>)number >> 1); // Should be
33                 ⇨ BitwiseHelpers.ShiftRight(number, 1);
34                 if (_equalityComparer.Equals(number, default))
35                 {
36                     break;
37                 }
38             }
39             return target;
40         }
41     }
42 }
```

./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> : LinksOperatorBase<TLink>,
8         ⇨ IConverter<Doublet<TLink>, TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ⇨ EqualityComparer<TLink>.Default;
12
13         private readonly ISpecificPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
14         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
15
16         public LinkToItsFrequencyNumberConveter(
17             ILinks<TLink> links,
18             ISpecificPropertyOperator<TLink, TLink> frequencyPropertyOperator,
19             IConverter<TLink> unaryNumberToAddressConverter)
20             : base(links)
21         {
22             _frequencyPropertyOperator = frequencyPropertyOperator;
23             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
24         }
25
26         public TLink Convert(Doublet<TLink> doublet)
27         {
28             var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
29             if (_equalityComparer.Equals(link, Links.Constants.Null))
30             {
31                 throw new ArgumentException($"Link with {doublet.Source} source and
32                     ⇨ {doublet.Target} target not found.", 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 = _unaryNumberToAddressConverter.Convert(frequencyNumber);
41             return number;
42         }
43     }
44 }
```

```

40     }
41 }

```

./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> : LinksOperatorBase<TLink>, IConverter<int,
8          ↪ 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) : base(links)
16         {
17             _unaryNumberPowersOf2 = new TLink[64];
18             _unaryNumberPowersOf2[0] = one;
19         }
20
21         public TLink Convert(int power)
22         {
23             if (power < 0 || power >= _unaryNumberPowersOf2.Length)
24             {
25                 throw new ArgumentOutOfRangeException(nameof(power));
26             }
27             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
28             {
29                 return _unaryNumberPowersOf2[power];
30             }
31             var previousPowerOf2 = Convert(power - 1);
32             var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
33             _unaryNumberPowersOf2[power] = powerOf2;
34             return powerOf2;
35         }
36     }
37 }

```

./Converters/UnaryNumberToAddressAddOperationConverter.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4
5  namespace Platform.Data.Doublets.Converters
6  {
7      public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
8          ↪ IConverter<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13         private Dictionary<TLink, TLink> _unaryToUInt64;
14         private readonly TLink _unaryOne;
15
16         public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
17             : base(links)
18         {
19             _unaryOne = unaryOne;
20             InitUnaryToUInt64();
21         }
22
23         private void InitUnaryToUInt64()
24         {
25             _unaryToUInt64 = new Dictionary<TLink, TLink>
26             {
27                 { _unaryOne, Integer<TLink>.One }
28             };
29             var unary = _unaryOne;
30             var number = Integer<TLink>.One;
31             for (var i = 1; i < 64; i++)
32             {
33                 _unaryToUInt64.Add(unary = Links.GetOrCreate(unary, unary), number =
34                     ↪ (Integer<TLink>)((Integer<TLink>)number * 2UL));
35             }
36         }
37
38         public TLink Convert(TLink unaryNumber)
39         {
40             if (_equalityComparer.Equals(unaryNumber, default))
41             {
42                 return default;
43             }
44             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
45             {

```

```

43         return Integer<TLink>.One;
44     }
45     var source = Links.GetSource(unaryNumber);
46     var target = Links.GetTarget(unaryNumber);
47     if (!_equalityComparer.Equals(source, target))
48     {
49         return _unaryToUInt64[unaryNumber];
50     }
51     else
52     {
53         var result = _unaryToUInt64[source];
54         TLink lastValue;
55         while (!_unaryToUInt64.TryGetValue(target, out lastValue))
56         {
57             source = Links.GetSource(target);
58             result = ArithmeticHelpers.Add(result, _unaryToUInt64[source]);
59             target = Links.GetTarget(target);
60         }
61         result = ArithmeticHelpers.Add(result, lastValue);
62         return result;
63     }
64 }
65 }
66 }

```

./Converters/UnaryNumberToAddressOrOperationConverter.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 UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
9      ↪ IConverter<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12         ↪ EqualityComparer<TLink>.Default;
13
14         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
15
16         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
17         ↪ TLink> powerOf2ToUnaryNumberConverter)
18         : base(links)
19         {
20             _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
21             for (int i = 0; i < CachedTypeInfo<TLink>.BitsLength; i++)
22             {
23                 _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
24             }
25
26             public TLink Convert(TLink sourceNumber)
27             {
28                 var source = sourceNumber;
29                 var target = Links.Constants.Null;
30                 while (!_equalityComparer.Equals(source, Links.Constants.Null))
31                 {
32                     if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
33                     {
34                         source = Links.Constants.Null;
35                     }
36                     else
37                     {
38                         powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
39                         source = Links.GetTarget(source);
40                     }
41                     target = (Integer<TLink>)((Integer<TLink>)target | 1UL << powerOf2Index); //
42                     ↪ MathHelpers.Or(target, MathHelpers.ShiftLeft(One, powerOf2Index))
43                 }
44                 return target;
45             }
46         }
47     }
48 }

```

./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 =
10         ↪ EqualityComparer<TLink>.Default;
11
12         public LinksCascadeDependenciesResolver(ILinks<TLink> links) : base(links) { }
13     }
14 }

```

```

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
26         //    ↳ операций в транзакциях
27         for (var i = (long)referencesCount - 1; i >= 0; i--)
28         {
29             if (!_equalityComparer.Equals(references[i], link))
30             {
31                 continue;
32             }
33             Links.Delete(references[i]);
34         }
35         ArrayPool.Free(references);
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> :
8          ↳ LinksUniquenessResolver<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↳ EqualityComparer<TLink>.Default;
12
13         public LinksCascadeUniquenessAndDependenciesResolver(ILinks<TLink> links) : base(links) { }
14
15         protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
16             ↳ newLinkAddress)
17         {
18             // TODO: Very similar to Merge (logic should be reused)
19             ulong referencesAsSourceCount = (Integer<TLink>)Links.Count(Constants.Any,
20                 ↳ oldLinkAddress, Constants.Any);
21             ulong referencesAsTargetCount = (Integer<TLink>)Links.Count(Constants.Any,
22                 ↳ Constants.Any, oldLinkAddress);
23             var references = ArrayPool.Allocate<TLink>((long)(referencesAsSourceCount +
24                 ↳ referencesAsTargetCount));
25             var referencesFiller = new ArrayFiller<TLink, TLink>(references, Constants.Continue);
26             Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, oldLinkAddress,
27                 ↳ Constants.Any);
28             Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, Constants.Any,
29                 ↳ oldLinkAddress);
30             for (ulong i = 0; i < referencesAsSourceCount; i++)
31             {
32                 var reference = references[i];
33                 if (!_equalityComparer.Equals(reference, oldLinkAddress))
34                 {
35                     Links.Update(reference, newLinkAddress, Links.GetTarget(reference));
36                 }
37             }
38             for (var i = (long)referencesAsSourceCount; i < references.Length; i++)
39             {
40                 var reference = references[i];
41                 if (!_equalityComparer.Equals(reference, oldLinkAddress))
42                 {
43                     Links.Update(reference, Links.GetSource(reference), newLinkAddress);
44                 }
45             }
46             ArrayPool.Free(references);
47             return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
48         }
49     }
50 }

```

./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) => Links.Count(restriction);
20
21         public virtual T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
22             ↳ Links.Each(handler, restrictions);
23
24         public virtual T Create() => Links.Create();
25
26         public virtual T Update(IList<T> restrictions) => Links.Update(restrictions);
27
28         public virtual void Delete(T link) => Links.Delete(link);
29     }

```

./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, ILinks<T>
9     {
10         public LinksCombinedConstants<T, T, int> Constants { get; }
11
12         public readonly ILinks<T> Links;
13
14         protected LinksDisposableDecoratorBase(ILinks<T> links)
15         {
16             Links = links;
17             Constants = links.Constants;
18         }
19
20         public virtual T Count(IList<T> restriction) => Links.Count(restriction);
21
22         public virtual T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
23             ↳ Links.Each(handler, restrictions);
24
25         public virtual T Create() => Links.Create();
26
27         public virtual T Update(IList<T> restrictions) => Links.Update(restrictions);
28
29         public virtual void Delete(T link) => Links.Delete(link);
30
31         protected override bool AllowMultipleDisposeCalls => true;
32
33         protected override void DisposeCore(bool manual, bool wasDisposed) =>
34             ↳ Disposable.TryDispose(Links);
35     }
36 }

```

./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 link to exist or to be
6     ↪ external (hybrid link's raw number).
7     public class LinksInnerReferenceValidator<T> : LinksDecoratorBase<T>
8     {
9         public LinksInnerReferenceValidator(ILinks<T> links) : base(links) { }
10
11         public override T Each(Func<IList<T>, T> handler, IList<T> restrictions)
12         {
13             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
14             return base.Each(handler, restrictions);
15         }
16
17         public override T Count(IList<T> restriction)
18         {
19             Links.EnsureInnerReferenceExists(restriction, nameof(restriction));
20             return base.Count(restriction);
21         }
22
23         public override T Update(IList<T> restrictions)
24         {
25             // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
26             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
27             return base.Update(restrictions);
28         }
29
30         public override void Delete(T link)
31         {
32             // TODO: Решить считать ли такое исключением, или лишь более конкретным требованием?
33             Links.EnsureLinkExists(link, nameof(link));
34             base.Delete(link);
35         }
36     }
37 }
```

./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 any specific
8     ↪ location inside ResizableDirectMemoryLinks.
9     /// This in turn will require to implement not a list of empty links, but a list of ranges to
10     ↪ store it more efficiently.
11     /// </remarks>
12     public class LinksNonExistentReferencesCreator<T> : LinksDecoratorBase<T>
13     {
14         public LinksNonExistentReferencesCreator(ILinks<T> links) : base(links) { }
15
16         public override T Update(IList<T> restrictions)
17         {
18             Links.EnsureCreated(restrictions[Constants.SourcePart],
19             ↪ restrictions[Constants.TargetPart]);
20             return base.Update(restrictions);
21         }
22     }
23 }
```

./Decorators/LinksNullToSelfReferenceResolver.cs

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

```

20         restrictions[Constants.TargetPart] =
21             ↪ _equalityComparer.Equals(restrictions[Constants.TargetPart], Constants.Null) ?
22             ↪ restrictions[Constants.IndexPart] : restrictions[Constants.TargetPart];
23     }
24 }

```

./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          public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
14          {
15              if (!_equalityComparer.Equals(Constants.Any, Constants.Itself)
16                  && ((restrictions.Count > Constants.IndexPart) &&
17                      ↪ _equalityComparer.Equals(restrictions[Constants.IndexPart], Constants.Itself))
18                  || ((restrictions.Count > Constants.SourcePart) &&
19                      ↪ _equalityComparer.Equals(restrictions[Constants.SourcePart], Constants.Itself))
20                  || ((restrictions.Count > Constants.TargetPart) &&
21                      ↪ _equalityComparer.Equals(restrictions[Constants.TargetPart], Constants.Itself))))
22              {
23                  return Constants.Continue;
24              }
25              return base.Each(handler, restrictions);
26          }
27
28          public override TLink Update(IList<TLink> restrictions)
29          {
30              restrictions[Constants.SourcePart] =
31                  ↪ _equalityComparer.Equals(restrictions[Constants.SourcePart], Constants.Itself) ?
32                  ↪ restrictions[Constants.IndexPart] : restrictions[Constants.SourcePart];
33              restrictions[Constants.TargetPart] =
34                  ↪ _equalityComparer.Equals(restrictions[Constants.TargetPart], Constants.Itself) ?
35                  ↪ restrictions[Constants.IndexPart] : restrictions[Constants.TargetPart];
36              return base.Update(restrictions);
37          }
38      }
39  }

```

./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 = Links.SearchOrDefault(restrictions[Constants.SourcePart],
15                 ↪ restrictions[Constants.TargetPart]);
16             if (_equalityComparer.Equals(newLinkAddress, default))
17             {
18                 return base.Update(restrictions);
19             }
20             return ResolveAddressChangeConflict(restrictions[Constants.IndexPart], newLinkAddress);
21         }
22
23         protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
24             ↪ newLinkAddress)
25         {
26             if (Links.Exists(oldLinkAddress))
27             {
28                 Delete(oldLinkAddress);
29             }
30             return newLinkAddress;
31         }
32     }
33 }

```

./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],
12                                     ↳ restrictions[Constants.TargetPart]);
13            return base.Update(restrictions);
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
10     ↳ взаимосвязей).
11     /// </summary>
12     /// <remarks>
13     /// Возможные оптимизации:
14     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
15     /// + меньше объём БД
16     /// - меньше производительность
17     /// - больше ограничение на количество связей в БД)
18     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
19     /// + меньше объём БД
20     /// - больше сложность
21     /// AVL - высота дерева может позволить точно расчитать размер дерева, нет необходимости в
22     ↳ SBT.
23     /// AVL дерево можно прошить.
24     /// Текущее теоретическое ограничение на размер связей - long.MaxValue
25     /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
26     ↳ (битовыми строками) - вариант матрицы (выстраиваемой лениво).
27     ///
28     /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
29     ↳ выбрасываться только при #if DEBUG
30     /// </remarks>
31     public class UInt64Links : LinksDisposableDecoratorBase<ulong>
32     {
33         public UInt64Links(ILinks<ulong> links) : base(links) { }
34
35         public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
36         {
37             this.EnsureLinkIsAnyOrExists(restrictions);
38             return Links.Each(handler, restrictions);
39         }
40
41         public override ulong Create() => Links.CreatePoint();
42
43         public override ulong Update(IList<ulong> restrictions)
44         {
45             if (restrictions.IsNullOrEmpty())
46             {
47                 return Constants.Null;
48             }
49             // TODO: Remove usages of these hacks (these should not be backwards compatible)
50             if (restrictions.Count == 2)
51             {
52                 return this.Merge(restrictions[0], restrictions[1]);
53             }
54             if (restrictions.Count == 4)

```



```

53     {
54         return this.UpdateOrCreateOrGet(restrictions[0], restrictions[1], restrictions[2],
55             ↳ restrictions[3]);
56     }
57     // TODO: Looks like this is a common type of exceptions linked with restrictions
58     ↳ support
59     if (restrictions.Count != 3)
60     {
61         throw new NotSupportedException();
62     }
63     var updatedLink = restrictions[Constants.IndexPart];
64     this.EnsureLinkExists(updatedLink, nameof(Constants.IndexPart));
65     var newSource = restrictions[Constants.SourcePart];
66     this.EnsureLinkIsItselfOrExists(newSource, nameof(Constants.SourcePart));
67     var newTarget = restrictions[Constants.TargetPart];
68     this.EnsureLinkIsItselfOrExists(newTarget, nameof(Constants.TargetPart));
69     var existedLink = Constants.Null;
70     if (newSource != Constants.Itself && newTarget != Constants.Itself)
71     {
72         existedLink = this.SearchOrDefault(newSource, newTarget);
73     }
74     if (existedLink == Constants.Null)
75     {
76         var before = Links.GetLink(updatedLink);
77         if (before[Constants.SourcePart] != newSource || before[Constants.TargetPart] !=
78             ↳ newTarget)
79         {
80             Links.Update(updatedLink, newSource == Constants.Itself ? updatedLink :
81                 ↳ newSource,
82                                     newTarget == Constants.Itself ? updatedLink :
83                                         ↳ newTarget);
84         }
85         return updatedLink;
86     }
87     else
88     {
89         // Replace one link with another (replaced link is deleted, children are updated
90         ↳ or deleted), it is actually merge operation
91         return this.Merge(updatedLink, existedLink);
92     }
93 }
94
95 /// <summary>Удаляет связь с указанным индексом.</summary>
96 /// <param name="link">Индекс удаляемой связи.</param>
97 public override void Delete(ulong link)
98 {
99     this.EnsureLinkExists(link);
100     Links.Update(link, Constants.Null, Constants.Null);
101     var referencesCount = Links.Count(Constants.Any, link);
102     if (referencesCount > 0)
103     {
104         var references = new ulong[referencesCount];
105         var referencesFiller = new ArrayFiller<ulong, ulong>(references,
106             ↳ Constants.Continue);
107         Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, link);
108         //references.Sort(); // TODO: Решить необходимо ли для корректного порядка отмены
109         ↳ 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             //else
117             // TODO: Определить почему здесь есть связи, которых не существует
118         }
119         Links.Delete(link);
120     }
121 }
122 }
123 }
124 }

```

./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 or Everything?
16  /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed by
    ↳ itself. But can cause creation (update from nothing) or deletion (update to nothing).
17  ///
18  /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
    ↳ DefaultUniLinksBase, that contains logic itself and can be implemented using both
    ↳ IDoubletLinks and ILinks.)
19  /// </remarks>
20  internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
21  {
22      private static readonly EqualityComparer<TLink> _equalityComparer =
    ↳ EqualityComparer<TLink>.Default;
23
24      public UniLinks(ILinks<TLink> links) : base(links) { }
25
26      private struct Transition
27      {
28          public IList<TLink> Before;
29          public IList<TLink> After;
30
31          public Transition(IList<TLink> before, IList<TLink> after)
32          {
33              Before = before;
34              After = after;
35          }
36      }
37
38      public static readonly TLink NullConstant = Use<LinksCombinedConstants<TLink>, TLink>,
    ↳ int>>.Single.Null;
39      public static readonly IReadOnlyList<TLink> NullLink = new ReadOnlyCollection<TLink>(new
    ↳ List<TLink> { NullConstant, NullConstant, NullConstant });
40
41      // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
    ↳ (Links-Expression)
42      public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
    ↳ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
    ↳ substitutedHandler)
43      {
44          ///List<Transition> transitions = null;
45          ///if (!restriction.IsNullOrEmpty())
46          ///{
47          ///    // Есть причина делать проход (чтение)
48          ///    if (matchedHandler != null)
49          ///    {
50          ///        if (!substitution.IsNullOrEmpty())
51          ///        {
52          ///            // restriction => { 0, 0, 0 } | { 0 } // Create
53          ///            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
    ↳ Create / Update
54          ///            // substitution => { 0, 0, 0 } | { 0 } // Delete
55          ///            transitions = new List<Transition>();
56          ///            if (Equals(substitution[Constants.IndexPart], Constants.Null))
57          ///            {
58          ///                // If index is Null, that means we always ignore every other value
    ↳ (they are also Null by definition)
59          ///                var matchDecision = matchedHandler(, NullLink);
60          ///                if (Equals(matchDecision, Constants.Break))
61          ///                    return false;
62          ///                if (!Equals(matchDecision, Constants.Skip))
63          ///                    transitions.Add(new Transition(matchedLink, newValue));
64          ///            }
65          ///            else
66          ///            {
67          ///                Func<T, bool> handler;
68          ///                handler = link =>
69          ///                {
70          ///                    var matchedLink = Memory.GetLinkValue(link);
71          ///                    var newValue = Memory.GetLinkValue(link);
72          ///                    newValue[Constants.IndexPart] = Constants.Itself;
73          ///                    newValue[Constants.SourcePart] =
    ↳ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
    ↳ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
74          ///                    newValue[Constants.TargetPart] =
    ↳ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
    ↳ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
75          ///                    var matchDecision = matchedHandler(matchedLink, newValue);
76          ///                    if (Equals(matchDecision, Constants.Break))
77          ///                        return false;
78          ///                    if (!Equals(matchDecision, Constants.Skip))
79          ///                        transitions.Add(new Transition(matchedLink, newValue));
80          ///                    return true;
81          ///                };
82          ///                if (!Memory.Each(handler, restriction))
83          ///                    return Constants.Break;
84          ///            }
85          ///        }
    }

```

```

86     // else
87     {
88         Func<T, bool> handler = link =>
89         {
90             var matchedLink = Memory.GetLinkValue(link);
91             var matchDecision = matchedHandler(matchedLink, matchedLink);
92             return !Equals(matchDecision, Constants.Break);
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" the state) // Each
140 //{
141 //    // No need to collect links to list
142 //    // Skip == Continue
143 //    // No need to check substitutedHandler
144 //    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
145 //        Constants.Break), restriction))
146 //        return Constants.Break;
147 //}
148 //else // Update
149 //{
150 //    //List<IList<T>>> matchedLinks = null;
151 //    if (matchedHandler != null)
152 //    {
153 //        matchedLinks = new List<IList<T>>>();
154 //        Func<T, bool> handler = link =>
155 //        {
156 //            var matchedLink = Memory.GetLinkValue(link);
157 //            var matchDecision = matchedHandler(matchedLink);
158 //            if (Equals(matchDecision, Constants.Break))
159 //                return false;
160 //            if (!Equals(matchDecision, Constants.Skip))
161 //                matchedLinks.Add(matchedLink);
162 //            return true;
163 //        };
164 //        if (!Memory.Each(handler, restriction))
165 //            return Constants.Break;
166 //    }
167 //    if (!matchedLinks.IsNullOrEmpty())
168 //    {
169 //        var totalMatchedLinks = matchedLinks.Count;
170 //        for (var i = 0; i < totalMatchedLinks; i++)
171 //        {

```

```

171         //         var matchedLink = matchedLinks[i];
172         //         if (substitutedHandler != null)
173         //         {
174         //             var newValue = new List<T>(); // TODO: Prepare value to update here
175         //             // TODO: Decide is it actually needed to use Before and After
176         //             ↪ substitution handling.
177         //             var substitutedDecision = substitutedHandler(matchedLink, newValue);
178         //             if (Equals(substitutedDecision, Constants.Break))
179         //                 return Constants.Break;
180         //             if (Equals(substitutedDecision, Constants.Continue))
181         //             {
182         //                 // Actual update here
183         //                 Memory.SetLinkValue(newValue);
184         //             }
185         //             if (Equals(substitutedDecision, Constants.Skip))
186         //             {
187         //                 // Cancel the update. TODO: decide use separate Cancel constant
188         //                 ↪ or Skip is enough?
189         //                 }
190         //             }
191         //         }
192         //     }
193     }
194     return Constants.Continue;
195 }
196
197 public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
198 ↪ matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
199 ↪ substitutionHandler)
200 {
201     if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
202     {
203         return Constants.Continue;
204     }
205     else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO: Check
206     ↪ if it is a correct condition
207     {
208         // Or it only applies to trigger without matchHandler.
209         throw new NotImplementedException();
210     }
211     else if (!substitution.IsNullOrEmpty()) // Creation
212     {
213         var before = ArrayPool<TLink>.Empty;
214         // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
215         ↪ (пройти мимо) или пустить (взять)?
216         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
217         ↪ Constants.Break))
218         {
219             return Constants.Break;
220         }
221         var after = (IList<TLink>)substitution.ToArray();
222         if (_equalityComparer.Equals(after[0], default))
223         {
224             var newLink = Links.Create();
225             after[0] = newLink;
226         }
227         if (substitution.Count == 1)
228         {
229             after = Links.GetLink(substitution[0]);
230         }
231         else if (substitution.Count == 3)
232         {
233             Links.Update(after);
234         }
235         else
236         {
237             throw new NotSupportedException();
238         }
239         if (matchHandler != null)
240         {
241             return substitutionHandler(before, after);
242         }
243         return Constants.Continue;
244     }
245     else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
246     {
247         if (patternOrCondition.Count == 1)
248         {
249             var linkToDelete = patternOrCondition[0];
250             var before = Links.GetLink(linkToDelete);
251             if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
252             ↪ Constants.Break))
253             {
254                 return Constants.Break;
255             }
256             var after = ArrayPool<TLink>.Empty;

```


./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 (IRefEqualityComparer)
8     /// 2x faster with comparer
9     /// </remarks>
10    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 DoubletComparer<T>();
16
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]
18        public bool Equals(Doublet<T> x, Doublet<T> y) => _equalityComparer.Equals(x.Source,
19            ↪ y.Source) && _equalityComparer.Equals(x.Target, y.Target);
20
21        [MethodImpl(MethodImplOptions.AggressiveInlining)]
22        public int GetHashCode(Doublet<T> obj) => unchecked(obj.Source.GetHashCode() << 15 ^
23            ↪ obj.Target.GetHashCode());
24    }
25 }
```

./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) => _equalityComparer.Equals(Source, other.Source) &&
23             ↪ _equalityComparer.Equals(Target, other.Target);
24     }
25 }
```

./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 => Math.Abs(Convert.ToInt64(To.Signed(Value)));
16
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,
27             ↪ CachedTypeInfo<T>.SignedVersion));
28
29         public Hybrid(object value, bool isExternal)
30         {
31             var signedType = CachedTypeInfo<T>.SignedVersion;
32             var signedValue = Convert.ChangeType(value, signedType);
33             var abs =
34                 ↪ typeof(MathHelpers).GetTypeInfo().GetMethod("Abs").MakeGenericMethod(signedType);
35         }
36     }
37 }
```

```

33     var negate = typeof(MathHelpers).GetTypeInfo().GetMethod("Negate").MakeGenericMethod(s
    ↪ ignedType);
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 }

```

```

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,
127             ↪ next))
128         {
129             //throw new Exception(string.Format("Невозможно выбрать путь, так как и Source
130             ↪ и Target совпадают с элементом пути {0}.", next));
131             return false;
132         }
133         if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
134             ↪ target))
135         {
136             //throw new Exception(string.Format("Невозможно продолжить путь через элемент
137             ↪ пути {0}", next));
138             return false;
139         }
140         current = next;
141     }
142     return true;
143 }
144
145 /// <remarks>
146 /// Может потребовать дополнительного стека для PathElement's при использовании
147 ↪ SequenceWalker.
148 /// </remarks>
149 public static TLink GetByKeyes<TLink>(this ILinks<TLink> links, TLink root, params int[]
150 ↪ path)
151 {
152     links.EnsureLinkExists(root, "root");
153     var currentLink = root;
154     for (var i = 0; i < path.Length; i++)
155     {
156         currentLink = links.GetLink(currentLink)[path[i]];
157     }
158     return currentLink;
159 }
160
161 public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink> links,
162 ↪ TLink root, ulong size, ulong index)
163 {
164     var constants = links.Constants;
165     var source = constants.SourcePart;
166     var target = constants.TargetPart;
167     if (!MathHelpers.IsPowerOfTwo(size))
168     {
169         throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
170             ↪ than powers of two are not supported.");
171     }
172     var path = new BitArray(BitConverter.GetBytes(index));
173     var length = BitwiseHelpers.GetLowestBitPosition(size);
174     links.EnsureLinkExists(root, "root");
175     var currentLink = root;
176     for (var i = length - 1; i >= 0; i--)
177     {
178         currentLink = links.GetLink(currentLink)[path[i] ? target : source];
179     }
180     return currentLink;
181 }
182
183 #endregion
184
185 /// <summary>
186 /// Возвращает индекс указанной связи.
187 /// </summary>
188 /// <param name="links">Хранилище связей.</param>
189 /// <param name="link">Связь представленная списком, состоящим из её адреса и
190 ↪ содержимого.</param>
191 /// <returns>Индекс начальной связи для указанной связи.</returns>
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

184 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
185     ↪ link[links.Constants.IndexPart];
186
187 /// <summary>
188 /// Возвращает индекс начальной (Source) связи для указанной связи.
189 /// </summary>
190 /// <param name="links">Хранилище связей.</param>
191 /// <param name="link">Индекс связи.</param>
192 /// <returns>Индекс начальной связи для указанной связи.</returns>
193 [MethodImpl(MethodImplOptions.AggressiveInlining)]
194 public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
195     ↪ links.GetLink(link)[links.Constants.SourcePart];
196
197 /// <summary>
198 /// Возвращает индекс начальной (Source) связи для указанной связи.
199 /// </summary>
200 /// <param name="links">Хранилище связей.</param>
201 /// <param name="link">Связь представленная списком, состоящим из её адреса и
202     ↪ содержимого.</param>
203 /// <returns>Индекс начальной связи для указанной связи.</returns>
204 [MethodImpl(MethodImplOptions.AggressiveInlining)]
205 public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
206     ↪ link[links.Constants.SourcePart];
207
208 /// <summary>
209 /// Возвращает индекс конечной (Target) связи для указанной связи.
210 /// </summary>
211 /// <param name="links">Хранилище связей.</param>
212 /// <param name="link">Индекс связи.</param>
213 /// <returns>Индекс конечной связи для указанной связи.</returns>
214 [MethodImpl(MethodImplOptions.AggressiveInlining)]
215 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
216     ↪ links.GetLink(link)[links.Constants.TargetPart];
217
218 /// <summary>
219 /// Возвращает индекс конечной (Target) связи для указанной связи.
220 /// </summary>
221 /// <param name="links">Хранилище связей.</param>
222 /// <param name="link">Связь представленная списком, состоящим из её адреса и
223     ↪ содержимого.</param>
224 /// <returns>Индекс конечной связи для указанной связи.</returns>
225 [MethodImpl(MethodImplOptions.AggressiveInlining)]
226 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
227     ↪ link[links.Constants.TargetPart];
228
229 /// <summary>
230 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик (handler)
231     ↪ для каждой подходящей связи.
232 /// </summary>
233 /// <param name="links">Хранилище связей.</param>
234 /// <param name="handler">Обработчик каждой подходящей связи.</param>
235 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение может
236     ↪ иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту, Any -
237     ↪ отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
238 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
239     ↪ случае.</returns>
240 [MethodImpl(MethodImplOptions.AggressiveInlining)]
241 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
242     ↪ handler, params TLink[] restrictions)
243     => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
244         ↪ links.Constants.Continue);
245
246 /// <summary>
247 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик (handler)
248     ↪ для каждой подходящей связи.
249 /// </summary>
250 /// <param name="links">Хранилище связей.</param>
251 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
252     ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
253     ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
254 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
255     ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
256     ↪ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
257 /// <param name="handler">Обработчик каждой подходящей связи.</param>
258 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
259     ↪ случае.</returns>
260 [MethodImpl(MethodImplOptions.AggressiveInlining)]
261 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
262     ↪ Func<TLink, bool> handler)
263 {
264     var constants = links.Constants;
265     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
266         ↪ constants.Break, constants.Any, source, target);
267 }

```

```

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 (проверить применяется
    ↪ ли эта логика)
371         max = Math.Min(max, (Integer<TLink>)constants.MaxPossibleIndex);
372         var createdLinks = new List<TLink>();
373         var equalityComparer = EqualityComparer<TLink>.Default;
374         TLink createdLink = creator();
375         while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
376         {
377             createdLinks.Add(createdLink);
378         }
379         for (var i = 0; i < createdLinks.Count; i++)
380         {
381             if (!nonExistentAddresses.Contains((Integer<TLink>)createdLinks[i]))
382             {
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 = (Integer<TLink>)links.Count(constants.Any, link,
    ↪ constants.Any);

```

```

397     var equalityComparer = EqualityComparer<TLink>.Default;
398     if (equalityComparer.Equals(values[constants.SourcePart], link))
399     {
400         referencesAsSource--;
401     }
402     ulong referencesAsTarget = (Integer<TLink>)links.Count(constants.Any, constants.Any,
403         ↪ link);
404     if (equalityComparer.Equals(values[constants.TargetPart], link))
405     {
406         referencesAsTarget--;
407     }
408     return referencesAsSource + referencesAsTarget;
409 }
410
411 /// <param name="links">Хранилище связей.</param>
412 [MethodImpl(MethodImplOptions.AggressiveInlining)]
413 public static bool DependenciesExist<TLink>(this ILinks<TLink> links, TLink link) =>
414     ↪ links.DependenciesCount(link) > 0;
415
416 /// <param name="links">Хранилище связей.</param>
417 [MethodImpl(MethodImplOptions.AggressiveInlining)]
418 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source, TLink
419     ↪ target)
420 {
421     var constants = links.Constants;
422     var values = links.GetLink(link);
423     var equalityComparer = EqualityComparer<TLink>.Default;
424     return equalityComparer.Equals(values[constants.SourcePart], source) &&
425         ↪ equalityComparer.Equals(values[constants.TargetPart], target);
426 }
427
428 /// <summary>
429 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
430 /// </summary>
431 /// <param name="links">Хранилище связей.</param>
432 /// <param name="source">Индекс связи, которая является началом для искомой связи.</param>
433 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
434 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
435     ↪ (концом).</returns>
436 [MethodImpl(MethodImplOptions.AggressiveInlining)]
437 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
438     ↪ target)
439 {
440     var constants = links.Constants;
441     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
442     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
443     return setter.Result;
444 }
445
446 /// <param name="links">Хранилище связей.</param>
447 [MethodImpl(MethodImplOptions.AggressiveInlining)]
448 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
449 {
450     var link = links.Create();
451     return links.Update(link, link, link);
452 }
453
454 /// <param name="links">Хранилище связей.</param>
455 [MethodImpl(MethodImplOptions.AggressiveInlining)]
456 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
457     ↪ target) => links.Update(links.Create(), source, target);
458
459 /// <summary>
460 /// Обновляет связь с указанными началом (Source) и концом (Target)
461 /// на связь с указанными началом (NewSource) и концом (NewTarget).
462 /// </summary>
463 /// <param name="links">Хранилище связей.</param>
464 /// <param name="link">Индекс обновляемой связи.</param>
465 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
466     ↪ выполняется обновление.</param>
467 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
468     ↪ выполняется обновление.</param>
469 /// <returns>Индекс обновлённой связи.</returns>
470 [MethodImpl(MethodImplOptions.AggressiveInlining)]
471 public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
472     ↪ TLink newTarget) => links.Update(new[] { link, newSource, newTarget });
473
474 /// <summary>
475 /// Обновляет связь с указанными началом (Source) и концом (Target)
476 /// на связь с указанными началом (NewSource) и концом (NewTarget).
477 /// </summary>
478 /// <param name="links">Хранилище связей.</param>
479 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение может
480     ↪ иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту, Itself -
481     ↪ требование установить ссылку на себя, 1..∞ конкретный адрес другой связи.</param>
482 /// <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],
481             ↪ restrictions[2], restrictions[3]);
482     }
483     else
484     {
485         return links.Update(restrictions);
486     }
487 }
488
489 /// <summary>
490 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи с
491 ↪ указанными Source (началом) и Target (концом).
492 /// </summary>
493 /// <param name="links">Хранилище связей.</param>
494 /// <param name="source">Индекс связи, которая является началом на создаваемой
495 ↪ связи.</param>
496 /// <param name="target">Индекс связи, которая является концом для создаваемой
497 ↪ связи.</param>
498 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
499 [MethodImpl(MethodImplOptions.AggressiveInlining)]
500 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
501 ↪ target)
502 {
503     var link = links.SearchOrDefault(source, target);
504     if (EqualityComparer<TLink>.Default.Equals(link, default))
505     {
506         link = links.CreateAndUpdate(source, target);
507     }
508     return link;
509 }
510
511 /// <summary>
512 /// Обновляет связь с указанными началом (Source) и концом (Target)
513 ↪ на связь с указанными началом (NewSource) и концом (NewTarget).
514 /// </summary>
515 /// <param name="links">Хранилище связей.</param>
516 /// <param name="source">Индекс связи, которая является началом обновляемой связи.</param>
517 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
518 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
519 ↪ выполняется обновление.</param>
520 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
521 ↪ выполняется обновление.</param>
522 /// <returns>Индекс обновлённой связи.</returns>
523 [MethodImpl(MethodImplOptions.AggressiveInlining)]
524 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
525 ↪ TLink target, TLink newSource, TLink newTarget)
526 {
527     var equalityComparer = EqualityComparer<TLink>.Default;
528     var link = links.SearchOrDefault(source, target);
529     if (equalityComparer.Equals(link, default))
530     {
531         return links.CreateAndUpdate(newSource, newTarget);
532     }
533     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
534 ↪ target))
535     {
536         return link;
537     }
538     return links.Update(link, newSource, newTarget);
539 }
540
541 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
542 /// <param name="links">Хранилище связей.</param>
543 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
544 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
545 [MethodImpl(MethodImplOptions.AggressiveInlining)]
546 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
547 ↪ target)
548 {
549     var link = links.SearchOrDefault(source, target);
550     if (!EqualityComparer<TLink>.Default.Equals(link, default))
551     {
552         links.Delete(link);
553         return link;
554     }
555     return default;
556 }

```

```

547 /// <summary>Удаляет несколько связей.</summary>
548 /// <param name="links">Хранилище связей.</param>
549 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
550 [MethodImpl(MethodImplOptions.AggressiveInlining)]
551 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
552 {
553     for (int i = 0; i < deletedLinks.Count; i++)
554     {
555         links.Delete(deletedLinks[i]);
556     }
557 }
558
559 // Replace one link with another (replaced link is deleted, children are updated or
560 //   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,
570     //   constants.Any);
571     ulong referencesAsTargetCount = (Integer<TLink>)links.Count(constants.Any,
572     //   constants.Any, linkIndex);
573     var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(linkIndex)) &&
574     //   referencesAsSourceCount == 1 && referencesAsTargetCount == 1;
575     if (!isStandalonePoint)
576     {
577         var totalReferences = referencesAsSourceCount + referencesAsTargetCount;
578         if (totalReferences > 0)
579         {
580             var references = ArrayPool.Allocate<TLink>((long)totalReferences);
581             var referencesFiller = new ArrayFiller<TLink, TLink>(references,
582             //   links.Constants.Continue);
583             links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any,
584             //   linkIndex, constants.Any);
585             links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any,
586             //   constants.Any, linkIndex);
587             for (ulong i = 0; i < referencesAsSourceCount; i++)
588             {
589                 var reference = references[i];
590                 if (equalityComparer.Equals(reference, linkIndex))
591                 {
592                     continue;
593                 }
594                 links.Update(reference, newLink, links.GetTarget(reference));
595             }
596             for (var i = (long)referencesAsSourceCount; i < references.Length; i++)
597             {
598                 var reference = references[i];
599                 if (equalityComparer.Equals(reference, linkIndex))
600                 {
601                     continue;
602                 }
603                 links.Update(reference, links.GetSource(reference), newLink);
604             }
605             ArrayPool.Free(references);
606         }
607     }
608     links.Delete(linkIndex);
609     return newLink;
610 }

```

./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 =
9         //   EqualityComparer<TLink>.Default;
10
11         private readonly TLink _frequencyMarker;
12         private readonly TLink _unaryOne;
13         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
14
15         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
16         //   IIncrementer<TLink> unaryNumberIncrementer)

```

```

15         : base(links)
16     {
17         _frequencyMarker = frequencyMarker;
18         _unaryOne = unaryOne;
19         _unaryNumberIncrementer = unaryNumberIncrementer;
20     }
21
22     public TLink Increment(TLink frequency)
23     {
24         if (_equalityComparer.Equals(frequency, default))
25         {
26             return Links.GetOrCreate(_unaryOne, _frequencyMarker);
27         }
28         var source = Links.GetSource(frequency);
29         var incrementedSource = _unaryNumberIncrementer.Increment(source);
30         return Links.GetOrCreate(incrementedSource, _frequencyMarker);
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> _frequencyPropertyOperator;
10         private readonly IIncrementer<TLink> _frequencyIncrementer;
11
12         public LinkFrequencyIncrementer(ILinks<TLink> links, ISpecificPropertyOperator<TLink,
13             ↳ TLink> frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
14             : base(links)
15         {
16             _frequencyPropertyOperator = frequencyPropertyOperator;
17             _frequencyIncrementer = frequencyIncrementer;
18
19             /// <remarks>Sequence itseft is not changed, only frequency of its doublets is
20             ↳ incremented.</remarks>
21         public IList<TLink> Increment(IList<TLink> sequence) // TODO: May be move to
22             ↳ ILinksExtensions or make SequenceDoubletsFrequencyIncrementer
23         {
24             for (var i = 1; i < sequence.Count; i++)
25             {
26                 Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
27             }
28             return sequence;
29         }
30
31         public void Increment(TLink link)
32         {
33             var previousFrequency = _frequencyPropertyOperator.Get(link);
34             var frequency = _frequencyIncrementer.Increment(previousFrequency);
35             _frequencyPropertyOperator.Set(link, frequency);
36         }
37     }
38 }

```

./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>, IIncrementer<TLink>
7      {
8          private static readonly EqualityComparer<TLink> _equalityComparer =
9             ↳ EqualityComparer<TLink>.Default;
10
11         private readonly TLink _unaryOne;
12
13         public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
14             ↳ _unaryOne = unaryOne;
15
16         public TLink Increment(TLink unaryNumber)
17         {
18             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
19             {
20                 return Links.GetOrCreate(_unaryOne, _unaryOne);
21             }
22             var source = Links.GetSource(unaryNumber);
23             var target = Links.GetTarget(unaryNumber);
24             if (_equalityComparer.Equals(source, target))
25             {
26                 return Links.GetOrCreate(unaryNumber, _unaryOne);
27             }
28         }
29     }
30 }

```



```

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, ILinks<TLink>,
        ↳ LinksCombinedConstants<TLink, TLink, int>>, ILinks<TLink>
6      {
7      }
8  }

```

./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>, IList<TLink>
15     {
16         public static readonly Link<TLink> Null = new Link<TLink>();
17
18         private static readonly LinksCombinedConstants<bool, TLink, int> _constants =
        ↳ Default<LinksCombinedConstants<bool, TLink, int>>.Instance;
19         private static readonly EqualityComparer<TLink> _equalityComparer =
        ↳ EqualityComparer<TLink>.Default;
20
21         private const int Length = 3;
22
23         public readonly TLink Index;
24         public readonly TLink Source;
25         public readonly TLink Target;
26
27         public Link(params TLink[] values)
28         {
29             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
        ↳ _constants.Null;
30             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
        ↳ _constants.Null;
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> &&
64         ↪ Equals((Link<TLink>)other);
65
66     public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
67         && _equalityComparer.Equals(Source, other.Source)
68         && _equalityComparer.Equals(Target, other.Target);
69
70     public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
71         ↪ {source}->{target})";
72
73     public static string ToString(TLink source, TLink target) => $"({source}->{target})";
74
75     public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
76
77     public static implicit operator Link<TLink> (TLink[] linkArray) => new
78         ↪ Link<TLink>(linkArray);
79
80     public TLink[] ToArray()
81     {
82         var array = new TLink[Length];
83         CopyTo(array, 0);
84         return array;
85     }
86
87     public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
88         ↪ ToString(Source, Target) : ToString(Index, Source, Target);
89
90     #region IList
91
92     public int Count => Length;
93
94     public bool IsReadOnly => true;
95
96     public TLink this[int index]
97     {
98         get
99         {
100             Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1), nameof(index));
101             if (index == _constants.IndexPart)
102             {
103                 return Index;
104             }
105             if (index == _constants.SourcePart)
106             {
107                 return Source;
108             }
109             if (index == _constants.TargetPart)
110             {
111                 return Target;
112             }
113             throw new NotSupportedException(); // Impossible path due to Ensure.ArgumentInRange
114         }
115         set => throw new NotSupportedException();
116     }
117
118     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
119
120     public IEnumerator<TLink> GetEnumerator()
121     {
122         yield return Index;
123         yield return Source;
124         yield return Target;
125     }
126
127     public void Add(TLink item) => throw new NotSupportedException();
128
129     public void Clear() => throw new NotSupportedException();
130
131     public bool Contains(TLink item) => IndexOf(item) >= 0;
132
133     public void CopyTo(TLink[] array, int arrayIndex)
134     {
135         Ensure.Always.ArgumentNotNull(array, nameof(array));
136         Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
137             ↪ nameof(arrayIndex));
138         if (arrayIndex + Length > array.Length)
139         {
140             throw new InvalidOperationException();
141         }
142         array[arrayIndex++] = Index;
143         array[arrayIndex++] = Source;
144         array[arrayIndex] = Target;
145     }
146
147     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) =>
6             ↪ Point<TLink>.IsFullPoint(link);
7         public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
8             ↪ Point<TLink>.IsPartialPoint(link);
9     }
10 }

```

./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 // <auto-generated>
3 //     Generated by the MSBuild WriteCodeFragment class.
4 // </auto-generated>
5 //-----
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
13     ↪ Class Library")]
14 [assembly: System.Reflection.AssemblyFileVersionAttribute("0.0.1.0")]
15 [assembly: System.Reflection.AssemblyInformationalVersionAttribute("0.0.1")]
16 [assembly: System.Reflection.AssemblyTitleAttribute("Platform.Data.Doublets")]
17 [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>,
8         ↪ IPropertyOperator<TLink, TLink, TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13         public DefaultLinkPropertyOperator(ILinks<TLink> links) : base(links)
14         {
15         }
16
17         public TLink GetValue(TLink @object, TLink property)
18         {
19             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 =>
35             ↪ link[Links.Constants.IndexPart]).ToList());
36         Links.GetOrCreate(objectProperty, value);
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>,
7          ↪ ISpecificPropertyOperator<TLink, TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↪ EqualityComparer<TLink>.Default;
11
12         private readonly TLink _frequencyPropertyMarker;
13         private readonly TLink _frequencyMarker;
14
15         public FrequencyPropertyOperator(ILinks<TLink> links, TLink frequencyPropertyMarker, TLink
16             ↪ frequencyMarker) : base(links)
17         {
18             _frequencyPropertyMarker = frequencyPropertyMarker;
19             _frequencyMarker = frequencyMarker;
20         }
21
22         public TLink Get(TLink link)
23         {
24             var property = Links.SearchOrDefault(link, _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, _frequencyMarker))
42                 {
43                     frequencyContainer = Links.GetIndex(candidate);
44                     return Links.Constants.Break;
45                 }
46                 return Links.Constants.Continue;
47             }, Links.Constants.Any, property, Links.Constants.Any);
48             return frequencyContainer;
49         }
50
51         private TLink GetFrequency(TLink container) => _equalityComparer.Equals(container,
52             ↪ default) ? default : Links.GetTarget(container);
53
54         public void Set(TLink link, TLink frequency)
55         {
56             var property = Links.GetOrCreate(link, _frequencyPropertyMarker);
57             var container = GetContainer(property);
58             if (_equalityComparer.Equals(container, default))
59             {
60                 Links.GetOrCreate(property, frequency);
61             }
62             else
63             {
64                 Links.Update(container, property, frequency);
65             }
66         }
67     }
68 }

```

```

61     }
62 }
63 }
64 }

```

./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, ILinks<TLink>
27     {
28         private static readonly EqualityComparer<TLink> _equalityComparer =
29             EqualityComparer<TLink>.Default;
30         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
31
32         /// <summary>Возвращает размер одной связи в байтах.</summary>
33         public static readonly int LinkSizeInBytes = StructureHelpers.SizeOf<Link>();
34
35         public static readonly int LinkHeaderSizeInBytes = StructureHelpers.SizeOf<LinkHeader>();
36
37         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
38
39         private struct Link
40         {
41             public static readonly int SourceOffset = Marshal.OffsetOf(typeof(Link),
42                 ↳ nameof(Source)).ToInt32();
43             public static readonly int TargetOffset = Marshal.OffsetOf(typeof(Link),
44                 ↳ nameof(Target)).ToInt32();
45             public static readonly int LeftAsSourceOffset = Marshal.OffsetOf(typeof(Link),
46                 ↳ nameof(LeftAsSource)).ToInt32();
47             public static readonly int RightAsSourceOffset = Marshal.OffsetOf(typeof(Link),
48                 ↳ nameof(RightAsSource)).ToInt32();
49             public static readonly int SizeAsSourceOffset = Marshal.OffsetOf(typeof(Link),
50                 ↳ nameof(SizeAsSource)).ToInt32();
51             public static readonly int LeftAsTargetOffset = Marshal.OffsetOf(typeof(Link),
52                 ↳ nameof(LeftAsTarget)).ToInt32();
53             public static readonly int RightAsTargetOffset = Marshal.OffsetOf(typeof(Link),
54                 ↳ nameof(RightAsTarget)).ToInt32();
55             public static readonly int SizeAsTargetOffset = Marshal.OffsetOf(typeof(Link),
56                 ↳ nameof(SizeAsTarget)).ToInt32();
57
58             public TLink Source;
59             public TLink Target;
60             public TLink LeftAsSource;
61             public TLink RightAsSource;
62             public TLink SizeAsSource;
63             public TLink LeftAsTarget;
64             public TLink RightAsTarget;
65             public TLink SizeAsTarget;
66
67             [MethodImpl(MethodImplOptions.AggressiveInlining)]
68             public static TLink GetSource(IntPtr pointer) => (pointer +
69                 ↳ SourceOffset).GetValue<TLink>();
70             [MethodImpl(MethodImplOptions.AggressiveInlining)]
71             public static TLink GetTarget(IntPtr pointer) => (pointer +
72                 ↳ TargetOffset).GetValue<TLink>();
73             [MethodImpl(MethodImplOptions.AggressiveInlining)]
74             public static TLink GetLeftAsSource(IntPtr pointer) => (pointer +
75                 ↳ LeftAsSourceOffset).GetValue<TLink>();
76             [MethodImpl(MethodImplOptions.AggressiveInlining)]
77             public static TLink GetRightAsSource(IntPtr pointer) => (pointer +
78                 ↳ RightAsSourceOffset).GetValue<TLink>();
79             [MethodImpl(MethodImplOptions.AggressiveInlining)]
80             public static TLink GetSizeAsSource(IntPtr pointer) => (pointer +
81                 ↳ SizeAsSourceOffset).GetValue<TLink>();
82             [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

69     public static TLink GetLeftAsTarget(IntPtr pointer) => (pointer +
    ↪ LeftAsTargetOffset).GetValue<TLink>();
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     public static TLink GetRightAsTarget(IntPtr pointer) => (pointer +
    ↪ RightAsTargetOffset).GetValue<TLink>();
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     public static TLink GetSizeAsTarget(IntPtr pointer) => (pointer +
    ↪ SizeAsTargetOffset).GetValue<TLink>();
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     public static void SetSource(IntPtr pointer, TLink value) => (pointer +
    ↪ SourceOffset).SetValue(value);
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     public static void SetTarget(IntPtr pointer, TLink value) => (pointer +
    ↪ TargetOffset).SetValue(value);
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     public static void SetLeftAsSource(IntPtr pointer, TLink value) => (pointer +
    ↪ LeftAsSourceOffset).SetValue(value);
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     public static void SetRightAsSource(IntPtr pointer, TLink value) => (pointer +
    ↪ RightAsSourceOffset).SetValue(value);
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     public static void SetSizeAsSource(IntPtr pointer, TLink value) => (pointer +
    ↪ SizeAsSourceOffset).SetValue(value);
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public static void SetLeftAsTarget(IntPtr pointer, TLink value) => (pointer +
    ↪ LeftAsTargetOffset).SetValue(value);
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     public static void SetRightAsTarget(IntPtr pointer, TLink value) => (pointer +
    ↪ RightAsTargetOffset).SetValue(value);
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     public static void SetSizeAsTarget(IntPtr pointer, TLink value) => (pointer +
    ↪ SizeAsTargetOffset).SetValue(value);
91 }
92
93 private struct LinksHeader
94 {
95     public static readonly int AllocatedLinksOffset =
    ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(AllocatedLinks)).ToInt32();
96     public static readonly int ReservedLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
    ↪ nameof(ReservedLinks)).ToInt32();
97     public static readonly int FreeLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
    ↪ nameof(FreeLinks)).ToInt32();
98     public static readonly int FirstFreeLinkOffset = Marshal.OffsetOf(typeof(LinksHeader),
    ↪ nameof(FirstFreeLink)).ToInt32();
99     public static readonly int FirstAsSourceOffset = Marshal.OffsetOf(typeof(LinksHeader),
    ↪ nameof(FirstAsSource)).ToInt32();
100    public static readonly int FirstAsTargetOffset = Marshal.OffsetOf(typeof(LinksHeader),
    ↪ nameof(FirstAsTarget)).ToInt32();
101    public static readonly int LastFreeLinkOffset = Marshal.OffsetOf(typeof(LinksHeader),
    ↪ nameof(LastFreeLink)).ToInt32();
102
103    public TLink AllocatedLinks;
104    public TLink ReservedLinks;
105    public TLink FreeLinks;
106    public TLink FirstFreeLink;
107    public TLink FirstAsSource;
108    public TLink FirstAsTarget;
109    public TLink LastFreeLink;
110    public TLink Reserved8;
111
112    [MethodImpl(MethodImplOptions.AggressiveInlining)]
113    public static TLink GetAllocatedLinks(IntPtr pointer) => (pointer +
    ↪ AllocatedLinksOffset).GetValue<TLink>();
114    [MethodImpl(MethodImplOptions.AggressiveInlining)]
115    public static TLink GetReservedLinks(IntPtr pointer) => (pointer +
    ↪ ReservedLinksOffset).GetValue<TLink>();
116    [MethodImpl(MethodImplOptions.AggressiveInlining)]
117    public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
    ↪ FreeLinksOffset).GetValue<TLink>();
118    [MethodImpl(MethodImplOptions.AggressiveInlining)]
119    public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
    ↪ FirstFreeLinkOffset).GetValue<TLink>();
120    [MethodImpl(MethodImplOptions.AggressiveInlining)]
121    public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
    ↪ FirstAsSourceOffset).GetValue<TLink>();
122    [MethodImpl(MethodImplOptions.AggressiveInlining)]
123    public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
    ↪ FirstAsTargetOffset).GetValue<TLink>();
124    [MethodImpl(MethodImplOptions.AggressiveInlining)]
125    public static TLink GetLastFreeLink(IntPtr pointer) => (pointer +
    ↪ LastFreeLinkOffset).GetValue<TLink>();
126
127    [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

128     public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) => pointer +
129         ↳ FirstAsSourceOffset;
130     public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
131         ↳ FirstAsTargetOffset;
132
133     [MethodImpl(MethodImplOptions.AggressiveInlining)]
134     public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
135         ↳ AllocatedLinksOffset).SetValue(value);
136     [MethodImpl(MethodImplOptions.AggressiveInlining)]
137     public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
138         ↳ ReservedLinksOffset).SetValue(value);
139     [MethodImpl(MethodImplOptions.AggressiveInlining)]
140     public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
141         ↳ FreeLinksOffset).SetValue(value);
142     [MethodImpl(MethodImplOptions.AggressiveInlining)]
143     public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
144         ↳ FirstFreeLinkOffset).SetValue(value);
145     [MethodImpl(MethodImplOptions.AggressiveInlining)]
146     public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
147         ↳ FirstAsSourceOffset).SetValue(value);
148     [MethodImpl(MethodImplOptions.AggressiveInlining)]
149     public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
150         ↳ FirstAsTargetOffset).SetValue(value);
151     [MethodImpl(MethodImplOptions.AggressiveInlining)]
152     public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
153         ↳ LastFreeLinkOffset).SetValue(value);
154 }
155
156 private readonly long _memoryReservationStep;
157
158 private readonly IResizableDirectMemory _memory;
159 private IntPtr _header;
160 private IntPtr _links;
161
162 private LinksTargetsTreeMethods _targetsTreeMethods;
163 private LinksSourcesTreeMethods _sourcesTreeMethods;
164
165 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой, нужно
166 ↳ использовать не список а дерево, так как так можно быстрее проверить на наличие связи
167 ↳ внутри
168 private UnusedLinksListMethods _unusedLinksListMethods;
169
170 /// <summary>
171 /// Возвращает общее число связей находящихся в хранилище.
172 /// </summary>
173 private TLink Total => Subtract(LinksHeader.GetAllocatedLinks(_header),
174     ↳ LinksHeader.GetFreeLinks(_header));
175
176 public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
177
178 public ResizableDirectMemoryLinks(string address)
179     : this(address, DefaultLinksSizeStep)
180 {
181 }
182
183 /// <summary>
184 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
185 ↳ минимальным шагом расширения базы данных.
186 /// </summary>
187 /// <param name="address">Полный путь к файлу базы данных.</param>
188 /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
189 ↳ байтах.</param>
190 public ResizableDirectMemoryLinks(string address, long memoryReservationStep)
191     : this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
192         ↳ memoryReservationStep)
193 {
194 }
195
196 public ResizableDirectMemoryLinks(IResizableDirectMemory memory)
197     : this(memory, DefaultLinksSizeStep)
198 {
199 }
200
201 public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
202     ↳ memoryReservationStep)
203 {
204     Constants = Default<LinksCombinedConstants<TLink, TLink, int>>.Instance;
205     _memory = memory;
206     _memoryReservationStep = memoryReservationStep;
207     if (memory.ReservedCapacity < memoryReservationStep)
208     {
209         memory.ReservedCapacity = memoryReservationStep;
210     }
211     SetPointers(_memory);
212     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks

```

```

200 _memory.UsedCapacity = ((long)(Integer<TLink>)LinksHeader.GetAllocatedLinks(_header) *
    ↳ LinkSizeInBytes) + LinkHeaderSizeInBytes;
201 // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
202 LinksHeader.SetReservedLinks(_header, (Integer<TLink>)((_memory.ReservedCapacity -
    ↳ LinkHeaderSizeInBytes) / LinkSizeInBytes));
203 }
204 [MethodImpl(MethodImplOptions.AggressiveInlining)]
205 public TLink Count(IList<TLink> restrictions)
206 {
207     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
208     if (restrictions.Count == 0)
209     {
210         return Total;
211     }
212     if (restrictions.Count == 1)
213     {
214         var index = restrictions[Constants.IndexPart];
215         if (_equalityComparer.Equals(index, Constants.Any))
216         {
217             return Total;
218         }
219         return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
220     }
221     if (restrictions.Count == 2)
222     {
223         var index = restrictions[Constants.IndexPart];
224         var value = restrictions[1];
225         if (_equalityComparer.Equals(index, Constants.Any))
226         {
227             if (_equalityComparer.Equals(value, Constants.Any))
228             {
229                 return Total; // Any - как отсутствие ограничения
230             }
231             return Add(_sourcesTreeMethods.CalculateReferences(value),
232                 ↳ _targetsTreeMethods.CalculateReferences(value));
233         }
234         else
235         {
236             if (!Exists(index))
237             {
238                 return Integer<TLink>.Zero;
239             }
240             if (_equalityComparer.Equals(value, Constants.Any))
241             {
242                 return Integer<TLink>.One;
243             }
244             var storedLinkValue = GetLinkUnsafe(index);
245             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
246                 ↳ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
247             {
248                 return Integer<TLink>.One;
249             }
250             return Integer<TLink>.Zero;
251         }
252     }
253     if (restrictions.Count == 3)
254     {
255         var index = restrictions[Constants.IndexPart];
256         var source = restrictions[Constants.SourcePart];
257         var target = restrictions[Constants.TargetPart];
258         if (_equalityComparer.Equals(index, Constants.Any))
259         {
260             if (_equalityComparer.Equals(source, Constants.Any) &&
261                 ↳ _equalityComparer.Equals(target, Constants.Any))
262             {
263                 return Total;
264             }
265             else if (_equalityComparer.Equals(source, Constants.Any))
266             {
267                 return _targetsTreeMethods.CalculateReferences(target);
268             }
269             else if (_equalityComparer.Equals(target, Constants.Any))
270             {
271                 return _sourcesTreeMethods.CalculateReferences(source);
272             }
273             else //if(source != Any && target != Any)
274             {
275                 // Эквивалент Exists(source, target) => Count(Any, 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

```



```

279     {
280         if (!Exists(index))
281         {
282             return Integer<TLink>.Zero;
283         }
284         if (_equalityComparer.Equals(source, Constants.Any) &&
285             ↪ _equalityComparer.Equals(target, Constants.Any))
286         {
287             return Integer<TLink>.One;
288         }
289         var storedLinkValue = GetLinkUnsafe(index);
290         if (!_equalityComparer.Equals(source, Constants.Any) &&
291             ↪ !_equalityComparer.Equals(target, Constants.Any))
292         {
293             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
294                 ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
295             {
296                 return Integer<TLink>.One;
297             }
298             return Integer<TLink>.Zero;
299         }
300         var value = default(TLink);
301         if (_equalityComparer.Equals(source, Constants.Any))
302         {
303             value = target;
304         }
305         if (_equalityComparer.Equals(target, Constants.Any))
306         {
307             value = source;
308         }
309         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
310             ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
311         {
312             return Integer<TLink>.One;
313         }
314         return Integer<TLink>.Zero;
315     }
316 }
317
318 [MethodImpl(MethodImplOptions.AggressiveInlining)]
319 public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
320 {
321     if (restrictions.Count == 0)
322     {
323         for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
324             ↪ (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
325             ↪ Increment(link))
326         {
327             if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
328                 ↪ Constants.Break))
329             {
330                 return Constants.Break;
331             }
332         }
333         return Constants.Continue;
334     }
335     if (restrictions.Count == 1)
336     {
337         var index = restrictions[Constants.IndexPart];
338         if (_equalityComparer.Equals(index, Constants.Any))
339         {
340             return Each(handler, ArrayPool<TLink>.Empty);
341         }
342         if (!Exists(index))
343         {
344             return Constants.Continue;
345         }
346         return handler(GetLinkStruct(index));
347     }
348     if (restrictions.Count == 2)
349     {
350         var index = restrictions[Constants.IndexPart];
351         var value = restrictions[1];
352         if (_equalityComparer.Equals(index, Constants.Any))
353         {
354             if (_equalityComparer.Equals(value, Constants.Any))
355             {
356                 return Each(handler, ArrayPool<TLink>.Empty);
357             }
358             if (_equalityComparer.Equals(Each(handler, new[] { index, value, Constants.Any
359                 ↪ }), Constants.Break))
360             {
361                 return Constants.Break;
362             }
363         }
364     }
365 }

```

```

358         return Constants.Break;
359     }
360     return Each(handler, new[] { index, Constants.Any, value });
361 }
362 else
363 {
364     if (!Exists(index))
365     {
366         return Constants.Continue;
367     }
368     if (_equalityComparer.Equals(value, Constants.Any))
369     {
370         return handler(GetLinkStruct(index));
371     }
372     var storedLinkValue = GetLinkUnsafe(index);
373     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
374         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
375     {
376         return handler(GetLinkStruct(index));
377     }
378     return Constants.Continue;
379 }
380 }
381 if (restrictions.Count == 3)
382 {
383     var index = restrictions[Constants.IndexPart];
384     var source = restrictions[Constants.SourcePart];
385     var target = restrictions[Constants.TargetPart];
386     if (_equalityComparer.Equals(index, Constants.Any))
387     {
388         if (_equalityComparer.Equals(source, Constants.Any) &&
389             ↪ _equalityComparer.Equals(target, Constants.Any))
390         {
391             return Each(handler, ArrayPool<TLink>.Empty);
392         }
393         else if (_equalityComparer.Equals(source, Constants.Any))
394         {
395             return _targetsTreeMethods.EachReference(target, handler);
396         }
397         else if (_equalityComparer.Equals(target, Constants.Any))
398         {
399             return _sourcesTreeMethods.EachReference(source, handler);
400         }
401         else //if(source != Any && target != Any)
402         {
403             var link = _sourcesTreeMethods.Search(source, target);
404             return _equalityComparer.Equals(link, Constants.Null) ? Constants.Continue
405                 ↪ : handler(GetLinkStruct(link));
406         }
407     }
408     else
409     {
410         if (!Exists(index))
411         {
412             return Constants.Continue;
413         }
414         if (_equalityComparer.Equals(source, Constants.Any) &&
415             ↪ _equalityComparer.Equals(target, Constants.Any))
416         {
417             return handler(GetLinkStruct(index));
418         }
419         var storedLinkValue = GetLinkUnsafe(index);
420         if (!_equalityComparer.Equals(source, Constants.Any) &&
421             ↪ !_equalityComparer.Equals(target, Constants.Any))
422         {
423             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
424                 _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
425             {
426                 return handler(GetLinkStruct(index));
427             }
428             return Constants.Continue;
429         }
430         var value = default(TLink);
431         if (_equalityComparer.Equals(source, Constants.Any))
432         {
433             value = target;
434         }
435         if (_equalityComparer.Equals(target, Constants.Any))
436         {
437             value = source;
438         }
439         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
440             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
441         {
442             return handler(GetLinkStruct(index));
443         }
444     }
445 }

```

```

440         return Constants.Continue;
441     }
442 }
443 throw new NotSupportedException("Другие размеры и способы ограничений не
↳ поддерживаются.");
444 }
445
446 /// <remarks>
447 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует в
↳ другом месте (но не в менеджере памяти, а в логике Links)
448 /// </remarks>
449 [MethodImpl(MethodImplOptions.AggressiveInlining)]
450 public TLink Update(ICollection<TLink> values)
451 {
452     var linkIndex = values[Constants.IndexPart];
453     var link = GetLinkUnsafe(linkIndex);
454     // Будет корректно работать только в том случае, если пространство выделенной связи
↳ предварительно заполнено нулями
455     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
456     {
457         _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
↳ linkIndex);
458     }
459     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
460     {
461         _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
↳ linkIndex);
462     }
463     Link.SetSource(link, values[Constants.SourcePart]);
464     Link.SetTarget(link, values[Constants.TargetPart]);
465     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
466     {
467         _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer(_header),
↳ linkIndex);
468     }
469     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
470     {
471         _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTargetPointer(_header),
↳ linkIndex);
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), Link.GetTarget(link));
481 }
482
483 [MethodImpl(MethodImplOptions.AggressiveInlining)]
484 private IntPtr GetLinkUnsafe(TLink linkIndex) => _links.GetElement(LinkSizeInBytes,
↳ linkIndex);
485
486 /// <remarks>
487 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
↳ пространство
488 /// </remarks>
489 public TLink Create()
490 {
491     var freeLink = LinksHeader.GetFirstFreeLink(_header);
492     if (!_equalityComparer.Equals(freeLink, Constants.Null))
493     {
494         _unusedLinksListMethods.Detach(freeLink);
495     }
496     else
497     {
498         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
↳ Constants.MaxPossibleIndex) > 0)
499         {
500             throw new
↳ LinksLimitReachedException((Integer<TLink>)Constants.MaxPossibleIndex);
501         }
502         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
↳ Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
503         {
504             _memory.ReservedCapacity += _memory.ReservationStep;
505             SetPointers(_memory);
506             LinksHeader.SetReservedLinks(_header,
↳ (Integer<TLink>)(_memory.ReservedCapacity / LinkSizeInBytes));
507         }
508         LinksHeader.SetAllocatedLinks(_header,
↳ Increment(LinksHeader.GetAllocatedLinks(_header)));
509         _memory.UsedCapacity += LinkSizeInBytes;
510         freeLink = LinksHeader.GetAllocatedLinks(_header);
511     }

```

```

512         return freeLink;
513     }
514
515     public void Delete(TLink link)
516     {
517         if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)
518         {
519             _unusedLinksListMethods.AttachAsFirst(link);
520         }
521         else if (_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
522         {
523             LinksHeader.SetAllocatedLinks(_header,
524                 ↳ Decrement(LinksHeader.GetAllocatedLinks(_header)));
525             ↳ memory.UsedCapacity -= LinkSizeInBytes;
526             // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
527             ↳ пока не дойдём до первой существующей связи
528             // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
529             while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
530                 ↳ Integer<TLink>.Zero) > 0) &&
531                 ↳ IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
532             {
533                 _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
534                 LinksHeader.SetAllocatedLinks(_header,
535                     ↳ Decrement(LinksHeader.GetAllocatedLinks(_header)));
536                 ↳ memory.UsedCapacity -= LinkSizeInBytes;
537             }
538         }
539     }
540
541     /// <remarks>
542     /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
543     ↳ адрес реально поменялся
544     ///
545     /// Указатель this.links может быть в том же месте,
546     /// так как 0-я связь не используется и имеет такой же размер как Header,
547     /// поэтому header размещается в том же месте, что и 0-я связь
548     /// </remarks>
549     private void SetPointers(IDirectMemory memory)
550     {
551         if (memory == null)
552         {
553             _links = IntPtr.Zero;
554             _header = _links;
555             _unusedLinksListMethods = null;
556             _targetsTreeMethods = null;
557             _unusedLinksListMethods = null;
558         }
559         else
560         {
561             _links = memory.Pointer;
562             _header = _links;
563             _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
564             _targetsTreeMethods = new LinksTargetsTreeMethods(this);
565             _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
566         }
567     }
568
569     [MethodImpl(MethodImplOptions.AggressiveInlining)]
570     private bool Exists(TLink link)
571     => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
572         && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
573         && !IsUnusedLink(link);
574
575     [MethodImpl(MethodImplOptions.AggressiveInlining)]
576     private bool IsUnusedLink(TLink link)
577     => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
578         || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)), Constants.Null)
579         && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)), Constants.Null));
580
581     #region DisposableBase
582     protected override bool AllowMultipleDisposeCalls => true;
583     protected override void DisposeCore(bool manual, bool wasDisposed)
584     {
585         if (!wasDisposed)
586         {
587             SetPointers(null);
588         }
589         Disposable.TryDispose(_memory);
590     }
591     #endregion
592 }

```

./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 : CircularDoublyLinkedListMethods<TLink>
10         {
11             private readonly IntPtr _links;
12             private readonly IntPtr _header;
13
14             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) => (_links.GetElement(LinkSizeInBytes,
31                 ↳ element) + Link.TargetOffset).GetValue<TLink>();
32
33             protected override TLink GetSize() => (_header +
34                 ↳ LinksHeader.FreeLinksOffset).GetValue<TLink>();
35
36             protected override void SetFirst(TLink element) => (_header +
37                 ↳ LinksHeader.FirstFreeLinkOffset).SetValue(element);
38
39             protected override void SetLast(TLink element) => (_header +
40                 ↳ LinksHeader.LastFreeLinkOffset).SetValue(element);
41
42             protected override void SetPrevious(TLink element, TLink previous) =>
43                 ↳ (_links.GetElement(LinkSizeInBytes, element) +
44                 ↳ Link.SourceOffset).SetValue(previous);
45
46             protected override void SetNext(TLink element, TLink next) =>
47                 ↳ (_links.GetElement(LinkSizeInBytes, element) + Link.TargetOffset).SetValue(next);
48
49             protected override void SetSize(TLink size) => (_header +
50                 ↳ LinksHeader.FreeLinksOffset).SetValue(size);
51         }
52     }
53 }
```

./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 : SizedAndThreadedAVLBalancedTreeMethods<TLink>
15         {
16             private readonly ResizableDirectMemoryLinks<TLink> _memory;
17             private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
18             protected readonly IntPtr Links;
19             protected readonly IntPtr Header;
20
21             protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
22             {
23                 Links = memory._links;
24                 Header = memory._header;
25                 _memory = memory;
26                 _constants = memory.Constants;
27             }
28
29             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             protected abstract TLink GetTreeRoot();
31
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             protected abstract TLink GetBasePartValue(TLink link);
34         }
35     }
36 }
```

```

34 public TLink this[TLink index]
35 {
36     get
37     {
38         var root = GetTreeRoot();
39         if (GreaterOrEqualThan(index, GetSize(root)))
40         {
41             return GetZero();
42         }
43         while (!EqualToZero(root))
44         {
45             var left = GetLeftOrDefault(root);
46             var leftSize = GetSizeOrZero(left);
47             if (LessThan(index, leftSize))
48             {
49                 root = left;
50                 continue;
51             }
52             if (IsEquals(index, leftSize))
53             {
54                 return root;
55             }
56             root = GetRightOrDefault(root);
57             index = Subtract(index, Increment(leftSize));
58         }
59         return GetZero(); // TODO: Impossible situation exception (only if tree
60             ↳ structure broken)
61     }
62 }
63
64 // TODO: Return indices range instead of references count
65 public TLink CalculateReferences(TLink link)
66 {
67     var root = GetTreeRoot();
68     var total = GetSize(root);
69     var totalRightIgnore = GetZero();
70     while (!EqualToZero(root))
71     {
72         var @base = GetBasePartValue(root);
73         if (LessOrEqualThan(@base, link))
74         {
75             root = GetRightOrDefault(root);
76         }
77         else
78         {
79             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
80             root = GetLeftOrDefault(root);
81         }
82     }
83     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, Increment(GetLeftSize(root)));
95             root = GetRightOrDefault(root);
96         }
97     }
98     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
99 }
100
101 public TLink EachReference(TLink link, Func<IList<TLink>, TLink> handler)
102 {
103     var root = GetTreeRoot();
104     if (EqualToZero(root))
105     {
106         return _constants.Continue;
107     }
108     TLink first = GetZero(), current = root;
109     while (!EqualToZero(current))
110     {
111         var @base = GetBasePartValue(current);
112         if (GreaterOrEqualThan(@base, link))
113         {
114             if (IsEquals(@base, link))
115             {
116                 first = current;
117             }
118             current = GetLeftOrDefault(current);
119         }
120     }

```

```

120     }
121     else
122     {
123         current = GetRightOrDefault(current);
124     }
125 }
126 if (!EqualToZero(first))
127 {
128     current = first;
129     while (true)
130     {
131         if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
132         {
133             return _constants.Break;
134         }
135         current = GetNext(current);
136         if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
137         {
138             break;
139         }
140     }
141 }
142 return _constants.Continue;
143 }
144
145 protected override void PrintNodeValue(TLink node, StringBuilder 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> memory)
159         : base(memory)
160     {
161     }
162
163     protected override IntPtr GetLeftPointer(TLink node) =>
164         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset;
165
166     protected override IntPtr GetRightPointer(TLink node) =>
167         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset;
168
169     protected override TLink GetLeftValue(TLink node) =>
170         ↪ (Links.GetElement(LinkSizeInBytes, node) +
171         ↪ Link.LeftAsSourceOffset).GetValue<TLink>();
172
173     protected override TLink GetRightValue(TLink node) =>
174         ↪ (Links.GetElement(LinkSizeInBytes, node) +
175         ↪ Link.RightAsSourceOffset).GetValue<TLink>();
176
177     protected override TLink GetSize(TLink node)
178     {
179         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
180             ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
181         return BitwiseHelpers.PartialRead(previousValue, 5, -5);
182     }
183
184     protected override void SetLeft(TLink node, TLink left) =>
185         ↪ (Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset).SetValue(left);
186
187     protected override void SetRight(TLink node, TLink right) =>
188         ↪ (Links.GetElement(LinkSizeInBytes, node) +
189         ↪ Link.RightAsSourceOffset).SetValue(right);
190
191     protected override void SetSize(TLink node, TLink size)
192     {
193         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
194             ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
195         (Links.GetElement(LinkSizeInBytes, node) +
196             ↪ Link.SizeAsSourceOffset).SetValue(BitwiseHelpers.PartialWrite(previousValue,
197             ↪ size, 5, -5));
198     }
199
200     protected override bool GetLeftIsChild(TLink node)
201     {
202         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
203             ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
204         return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 4, 1);
205     }
206 }

```

```

190 }
191
192 protected override void SetLeftIsChild(TLink node, bool value)
193 {
194     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
195         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
196     var modified = BitwiseHelpers.PartialWrite(previousValue,
197         ↪ (TLink)(Integer<TLink>)value, 4, 1);
198     (Links.GetElement(LinkSizeInBytes, node) +
199         ↪ Link.SizeAsSourceOffset).SetValue(modified);
200 }
201
202 protected override bool GetRightIsChild(TLink node)
203 {
204     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
205         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
206     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 3, 1);
207 }
208
209 protected override void SetRightIsChild(TLink node, bool value)
210 {
211     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
212         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
213     var modified = BitwiseHelpers.PartialWrite(previousValue,
214         ↪ (TLink)(Integer<TLink>)value, 3, 1);
215     (Links.GetElement(LinkSizeInBytes, node) +
216         ↪ Link.SizeAsSourceOffset).SetValue(modified);
217 }
218
219 protected override sbyte GetBalance(TLink node)
220 {
221     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
222         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
223     var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 0, 3);
224     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 | 124
225         ↪ : value & 3);
226     return unpackedValue;
227 }
228
229 protected override void SetBalance(TLink node, sbyte value)
230 {
231     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
232         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
233     var packagedValue = (TLink)(Integer<TLink>)(((byte)value >> 5) & 4) | value & 3);
234     var modified = BitwiseHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
235     (Links.GetElement(LinkSizeInBytes, node) +
236         ↪ Link.SizeAsSourceOffset).SetValue(modified);
237 }
238
239 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
240 {
241     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
242         ↪ Link.SourceOffset).GetValue<TLink>();
243     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
244         ↪ Link.SourceOffset).GetValue<TLink>();
245     return LessThan(firstSource, secondSource) ||
246         (Equals(firstSource, secondSource) &&
247         ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
248         ↪ Link.TargetOffset).GetValue<TLink>(),
249         ↪ (Links.GetElement(LinkSizeInBytes, second) +
250         ↪ Link.TargetOffset).GetValue<TLink>()));
251 }
252
253 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
254 {
255     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
256         ↪ Link.SourceOffset).GetValue<TLink>();
257     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
258         ↪ Link.SourceOffset).GetValue<TLink>();
259     return GreaterThan(firstSource, secondSource) ||
260         (Equals(firstSource, secondSource) &&
261         ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
262         ↪ Link.TargetOffset).GetValue<TLink>(),
263         ↪ (Links.GetElement(LinkSizeInBytes, second) +
264         ↪ Link.TargetOffset).GetValue<TLink>()));
265 }
266
267 protected override TLink GetTreeRoot() => (Header +
268     ↪ LinksHeader.FirstAsSourceOffset).GetValue<TLink>();
269
270 protected override TLink GetBasePartValue(TLink link) =>
271     ↪ (Links.GetElement(LinkSizeInBytes, link) + Link.SourceOffset).GetValue<TLink>();
272
273 /// <summary>
274

```



```

249     /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
250     ↪ (концом)
251     /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
252     /// </summary>
253     /// <param name="source">Индекс связи, которая является началом на искомой
254     ↪ связи.</param>
255     /// <param name="target">Индекс связи, которая является концом на искомой
256     ↪ связи.</param>
257     /// <returns>Индекс искомой связи.</returns>
258     public TLink Search(TLink source, TLink target)
259     {
260         var root = GetTreeRoot();
261         while (!EqualToZero(root))
262         {
263             var rootSource = (Links.GetElement(LinkSizeInBytes, root) +
264             ↪ Link.SourceOffset).GetValue<TLink>();
265             var rootTarget = (Links.GetElement(LinkSizeInBytes, root) +
266             ↪ Link.TargetOffset).GetValue<TLink>();
267             if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
268             ↪ node.Key < root.Key
269             {
270                 root = GetLeftOrDefault(root);
271             }
272             else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
273             ↪ node.Key > root.Key
274             {
275                 root = GetRightOrDefault(root);
276             }
277             else // node.Key == root.Key
278             {
279                 return root;
280             }
281         }
282         return GetZero();
283     }
284
285     [MethodImpl(MethodImplOptions.AggressiveInlining)]
286     private bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget, TLink
287     ↪ secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
288     ↪ (IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
289
290     [MethodImpl(MethodImplOptions.AggressiveInlining)]
291     private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget, TLink
292     ↪ secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
293     ↪ (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
294
295     }
296
297     private class LinksTargetsTreeMethods : LinksTreeMethodsBase
298     {
299         public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
300         : base(memory)
301         {
302         }
303
304         protected override IntPtr GetLeftPointer(TLink node) =>
305         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset;
306
307         protected override IntPtr GetRightPointer(TLink node) =>
308         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset;
309
310         protected override TLink GetLeftValue(TLink node) =>
311         ↪ (Links.GetElement(LinkSizeInBytes, node) +
312         ↪ Link.LeftAsTargetOffset).GetValue<TLink>();
313
314         protected override TLink GetRightValue(TLink node) =>
315         ↪ (Links.GetElement(LinkSizeInBytes, node) +
316         ↪ Link.RightAsTargetOffset).GetValue<TLink>();
317
318         protected override TLink GetSize(TLink node)
319         {
320             var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
321             ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
322             return BitwiseHelpers.PartialRead(previousValue, 5, -5);
323         }
324
325         protected override void SetLeft(TLink node, TLink left) =>
326         ↪ (Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset).SetValue(left);
327
328         protected override void SetRight(TLink node, TLink right) =>
329         ↪ (Links.GetElement(LinkSizeInBytes, node) +
330         ↪ Link.RightAsTargetOffset).SetValue(right);
331
332         protected override void SetSize(TLink node, TLink size)
333         {
334         }
335     }

```

```

312     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
313         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
314     (Links.GetElement(LinkSizeInBytes, node) +
315         ↪ Link.SizeAsTargetOffset).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 (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 4, 1);
324 }
325
326 protected override void SetLeftIsChild(TLink node, bool value)
327 {
328     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
329         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
330     var modified = BitwiseHelpers.PartialWrite(previousValue,
331         ↪ (TLink)(Integer<TLink>)value, 4, 1);
332     (Links.GetElement(LinkSizeInBytes, node) +
333         ↪ Link.SizeAsTargetOffset).SetValue(modified);
334 }
335
336 protected override bool GetRightIsChild(TLink node)
337 {
338     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
339         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
340     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 3, 1);
341 }
342
343 protected override void SetRightIsChild(TLink node, bool value)
344 {
345     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
346         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
347     var modified = BitwiseHelpers.PartialWrite(previousValue,
348         ↪ (TLink)(Integer<TLink>)value, 3, 1);
349     (Links.GetElement(LinkSizeInBytes, node) +
350         ↪ Link.SizeAsTargetOffset).SetValue(modified);
351 }
352
353 protected override sbyte GetBalance(TLink node)
354 {
355     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
356         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
357     var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 0, 3);
358     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 | 124
359         ↪ : value & 3);
360     return unpackedValue;
361 }
362
363 protected override void SetBalance(TLink node, sbyte value)
364 {
365     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
366         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
367     var packagedValue = (TLink)(Integer<TLink>)(((byte)value >> 5) & 4) | value & 3;
368     var modified = BitwiseHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
369     (Links.GetElement(LinkSizeInBytes, node) +
370         ↪ Link.SizeAsTargetOffset).SetValue(modified);
371 }
372
373 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
374 {
375     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
376         ↪ Link.TargetOffset).GetValue<TLink>();
377     var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
378         ↪ Link.TargetOffset).GetValue<TLink>();
379     return LessThan(firstTarget, secondTarget) ||
380         (Equals(firstTarget, secondTarget) &&
381         ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
382         ↪ Link.SourceOffset).GetValue<TLink>(),
383         ↪ (Links.GetElement(LinkSizeInBytes, second) +
384         ↪ Link.SourceOffset).GetValue<TLink>()));
385 }
386
387 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
388 {
389     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
390         ↪ Link.TargetOffset).GetValue<TLink>();
391     var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
392         ↪ Link.TargetOffset).GetValue<TLink>();
393     return GreaterThan(firstTarget, secondTarget) ||

```

```

371         (IsEquals(firstTarget, secondTarget) &&
            GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
            ↪ Link.SourceOffset).GetValue<TLink>(),
            ↪ (Links.GetElement(LinkSizeInBytes, second) +
            ↪ Link.SourceOffset).GetValue<TLink>()));
372     }
373
374     protected override TLink GetTreeRoot() => (Header +
        ↪ LinksHeader.FirstAsTargetOffset).GetValue<TLink>();
375
376     protected override TLink GetBasePartValue(TLink link) =>
        ↪ (Links.GetElement(LinkSizeInBytes, link) + Link.TargetOffset).GetValue<TLink>();
377 }
378 }
379 }

```

./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 : DisposableBase, ILinks<id>
23      {
24          /// <summary>Возвращает размер одной связи в байтах.</summary>
25          /// <remarks>
26          /// Используется только во вне класса, не рекомендуется использовать внутри.
27          /// Так как во вне не обязательно будет доступен unsafe C#.
28          /// </remarks>
29          public static readonly int LinkSizeInBytes = sizeof(Link);
30
31          public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
32
33          private struct Link
34          {
35              public id Source;
36              public id Target;
37              public id LeftAsSource;
38              public id RightAsSource;
39              public id SizeAsSource;
40              public id LeftAsTarget;
41              public id RightAsTarget;
42              public id SizeAsTarget;
43          }
44
45          private struct LinksHeader
46          {
47              public id AllocatedLinks;
48              public id ReservedLinks;
49              public id FreeLinks;
50              public id FirstFreeLink;
51              public id FirstAsSource;
52              public id FirstAsTarget;
53              public id LastFreeLink;
54              public id Reserved8;
55          }
56
57          private readonly long _memoryReservationStep;
58
59          private readonly IResizableDirectMemory _memory;
60          private LinksHeader* _header;
61          private Link* _links;
62
63          private LinksTargetsTreeMethods _targetsTreeMethods;
64          private LinksSourcesTreeMethods _sourcesTreeMethods;
65
66          // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой, нужно
67          ↪ использовать не список а дерево, так как так можно быстрее проверить на наличие связи
68          ↪ внутри
69          private UnusedLinksListMethods _unusedLinksListMethods;
70
71          /// <summary>
72          /// Возвращает общее число связей находящихся в хранилище.
73          /// </summary>
74          private id Total => _header->AllocatedLinks - _header->FreeLinks;

```

```

73 // TODO: Дать возможность переопределять в конструкторе
74 public LinksCombinedConstants<id, id, int> Constants { get; }
75
76
77 public UInt64ResizableDirectMemoryLinks(string address) : this(address,
    ↳ DefaultLinksSizeStep) { }
78
79 /// <summary>
80 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, 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 базы данных в
    ↳ байтах.</param>
84 public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
    ↳ this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
    ↳ memoryReservationStep) { }
85
86 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
    ↳ DefaultLinksSizeStep) { }
87
88 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep)
89 {
90     Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
91     _memory = memory;
92     _memoryReservationStep = memoryReservationStep;
93     if (memory.ReservedCapacity < memoryReservationStep)
94     {
95         memory.ReservedCapacity = memoryReservationStep;
96     }
97     SetPointers(_memory);
98     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
99     _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) +
    ↳ sizeof(LinksHeader);
100    // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
101    _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(LinksHeader)) /
    ↳ sizeof(Link));
102 }
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 public id Count(IList<id> restrictions)
106 {
107     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
108     if (restrictions.Count == 0)
109     {
110         return Total;
111     }
112     if (restrictions.Count == 1)
113     {
114         var index = restrictions[Constants.IndexPart];
115         if (index == Constants.Any)
116         {
117             return Total;
118         }
119         return Exists(index) ? 1UL : 0UL;
120     }
121     if (restrictions.Count == 2)
122     {
123         var index = restrictions[Constants.IndexPart];
124         var value = restrictions[1];
125         if (index == Constants.Any)
126         {
127             if (value == Constants.Any)
128             {
129                 return Total; // Any - как отсутствие ограничения
130             }
131             return _sourcesTreeMethods.CalculateReferences(value)
132                 + _targetsTreeMethods.CalculateReferences(value);
133         }
134         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, source, target) > 0
175             var link = _sourcesTreeMethods.Search(source, target);
176             return link == Constants.Null ? OUL : 1UL;
177         }
178     }
179     else
180     {
181         if (!Exists(index))
182         {
183             return 0;
184         }
185         if (source == Constants.Any && target == Constants.Any)
186         {
187             return 1;
188         }
189         var storedLinkValue = GetLinkUnsafe(index);
190         if (source != Constants.Any && target != Constants.Any)
191         {
192             if (storedLinkValue->Source == source &&
193                 storedLinkValue->Target == target)
194             {
195                 return 1;
196             }
197             return 0;
198         }
199         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 }) == Constants.Break)
260         {
261             return Constants.Break;
262         }
263         return Each(handler, new[] { index, Constants.Any, value });
264     }
265     else
266     {
267         if (!Exists(index))
268         {
269             return Constants.Continue;
270         }
271         if (value == Constants.Any)
272         {
273             return handler(GetLinkStruct(index));
274         }
275         var storedLinkValue = GetLinkUnsafe(index);
276         if (storedLinkValue->Source == value ||
277             storedLinkValue->Target == value)
278         {
279             return handler(GetLinkStruct(index));
280         }
281         return Constants.Continue;
282     }
283 }
284 if (restrictions.Count == 3)
285 {
286     var index = restrictions[Constants.IndexPart];
287     var source = restrictions[Constants.SourcePart];
288     var target = restrictions[Constants.TargetPart];
289     if (index == Constants.Any)
290     {
291         if (source == Constants.Any && target == Constants.Any)
292         {
293             return Each(handler, ArrayPool<ulong>.Empty);
294         }
295         else if (source == Constants.Any)
296         {
297             return _targetsTreeMethods.EachReference(target, handler);
298         }
299         else if (target == Constants.Any)
300         {
301             return _sourcesTreeMethods.EachReference(source, handler);
302         }
303         else //if(source != Any && target != Any)
304         {
305             var link = _sourcesTreeMethods.Search(source, target);
306             return link == Constants.Null ? Constants.Continue :
307                 ↪ handler(GetLinkStruct(link));
308         }
309     }
310     else
311     {
312         if (!Exists(index))
313         {
314             return Constants.Continue;
315         }
316         if (source == Constants.Any && target == Constants.Any)
317         {
318             return handler(GetLinkStruct(index));
319         }
320         var storedLinkValue = GetLinkUnsafe(index);
321         if (source != Constants.Any && target != Constants.Any)

```

```

321     {
322         if (storedLinkValue->Source == source &&
323             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 ограничений не
    ↳ поддерживаются.");
347 }
348
349 /// <remarks>
350 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует в
    ↳ другом месте (но не в менеджере памяти, а в логике Links)
351 /// </remarks>
352 [MethodImpl(MethodImplOptions.AggressiveInlining)]
353 public id Update(IList<id> values)
354 {
355     var linkIndex = values[Constants.IndexPart];
356     var link = GetLinkUnsafe(linkIndex);
357     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
358     if (link->Source != Constants.Null)
359     {
360         _sourcesTreeMethods.Detach(new IntPtr(&_header->FirstAsSource), linkIndex);
361     }
362     if (link->Target != Constants.Null)
363     {
364         _targetsTreeMethods.Detach(new IntPtr(&_header->FirstAsTarget), linkIndex);
365     }
366 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
367     var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
368     var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
369     if (leftTreeSize != rightTreeSize)
370     {
371         throw new Exception("One of the trees is broken.");
372     }
373 #endif
374     link->Source = values[Constants.SourcePart];
375     link->Target = values[Constants.TargetPart];
376     if (link->Source != Constants.Null)
377     {
378         _sourcesTreeMethods.Attach(new IntPtr(&_header->FirstAsSource), linkIndex);
379     }
380     if (link->Target != Constants.Null)
381     {
382         _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
383     }
384 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
385     leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
386     rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
387     if (leftTreeSize != rightTreeSize)
388     {
389         throw new Exception("One of the trees is broken.");
390     }
391 #endif
392     return linkIndex;
393 }
394
395 [MethodImpl(MethodImplOptions.AggressiveInlining)]
396 private IList<id> GetLinkStruct(id linkIndex)
397 {
398     var link = GetLinkUnsafe(linkIndex);
399     return new UInt64Link(linkIndex, link->Source, link->Target);
400 }
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 не заполняет
407   ↳ 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 LinksLimitReachedException(Constants.MaxPossibleIndex);
421         }
422         if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
423         {
424             _memory.ReservedCapacity += _memory.ReservationStep;
425             SetPointers(_memory);
426             _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
427         }
428         _header->AllocatedLinks++;
429         _memory.UsedCapacity += sizeof(Link);
430         freeLink = _header->AllocatedLinks;
431     }
432     return freeLink;
433 }
434 public void Delete(id link)
435 {
436     if (link < _header->AllocatedLinks)
437     {
438         _unusedLinksListMethods.AttachAsFirst(link);
439     }
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 файла, до тех пор,
445         // ↳ пока не дойдём до первой существующей связи
446         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
447         while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
448         {
449             _unusedLinksListMethods.Detach(_header->AllocatedLinks);
450             _header->AllocatedLinks--;
451             _memory.UsedCapacity -= sizeof(Link);
452         }
453     }
454 }
455 /// <remarks>
456 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
457   ↳ адрес реально поменялся
458 ///
459 /// Указатель this.links может быть в том же месте,
460 /// так как 0-я связь не используется и имеет такой же размер как Header,
461 /// поэтому header размещается в том же месте, что и 0-я связь
462 /// </remarks>
463 private void SetPointers(IResizableDirectMemory memory)
464 {
465     if (memory == null)
466     {
467         _header = null;
468         _links = null;
469         _unusedLinksListMethods = null;
470         _targetsTreeMethods = null;
471         _unusedLinksListMethods = null;
472     }
473     else
474     {
475         _header = (LinksHeader*)(void*)memory.Pointer;
476         _links = (Link*)(void*)memory.Pointer;
477         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
478         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
479         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
480     }
481 }
482 [MethodImpl(MethodImplOptions.AggressiveInlining)]
483 private bool Exists(id link) => link >= Constants.MinPossibleIndex && link <=
484   ↳ _header->AllocatedLinks && !IsUnusedLink(link);
485 [MethodImpl(MethodImplOptions.AggressiveInlining)]
486 private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
487   || (_links[link].SizeAsSource == Constants.Null &&
488   ↳ _links[link].Source != Constants.Null);

```



```

488
489     #region Disposable
490
491     protected override bool AllowMultipleDisposeCalls => true;
492
493     protected override void DisposeCore(bool manual, bool wasDisposed)
494     {
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 : CircularDoublyLinkedListMethods<ulong>
8          {
9              private readonly Link* _links;
10             private readonly LinksHeader* _header;
11
12             public UnusedLinksListMethods(Link* links, LinksHeader* header)
13             {
14                 _links = links;
15                 _header = header;
16             }
17
18             protected override ulong GetFirst() => _header->FirstFreeLink;
19
20             protected override ulong GetLast() => _header->LastFreeLink;
21
22             protected override ulong GetPrevious(ulong element) => _links[element].Source;
23
24             protected override ulong GetNext(ulong element) => _links[element].Target;
25
26             protected override ulong GetSize() => _header->FreeLinks;
27
28             protected override void SetFirst(ulong element) => _header->FirstFreeLink = element;
29
30             protected override void SetLast(ulong element) => _header->LastFreeLink = element;
31
32             protected override void SetPrevious(ulong element, ulong previous) =>
33                 ↪ _links[element].Source = previous;
34
35             protected override void SetNext(ulong element, ulong next) => _links[element].Target =
36                 ↪ next;
37
38             protected override void SetSize(ulong size) => _header->FreeLinks = size;
39         }
40     }
41 }

```

./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 : SizedAndThreadedAVLBalancedTreeMethods<ulong>
13         {
14             private readonly UInt64ResizableDirectMemoryLinks _memory;
15             private readonly LinksCombinedConstants<ulong, ulong, int> _constants;
16             protected readonly Link* Links;
17             protected readonly LinksHeader* Header;
18
19             protected LinksTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory)
20             {
21                 Links = memory._links;
22                 Header = memory._header;
23                 _memory = memory;
24                 _constants = memory.Constants;
25             }
26
27             [MethodImpl(MethodImplOptions.AggressiveInlining)]
28             protected abstract ulong GetTreeRoot();

```

```

29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 protected abstract ulong GetBasePartValue(ulong link);
31
32 public ulong this[ulong index]
33 {
34     get
35     {
36         var root = GetTreeRoot();
37         if (index >= GetSize(root))
38         {
39             return 0;
40         }
41         while (root != 0)
42         {
43             var left = GetLeftOrDefault(root);
44             var leftSize = GetSizeOrZero(left);
45             if (index < leftSize)
46             {
47                 root = left;
48                 continue;
49             }
50             if (index == leftSize)
51             {
52                 return root;
53             }
54             root = GetRightOrDefault(root);
55             index -= leftSize + 1;
56         }
57         return 0; // TODO: Impossible situation exception (only if tree structure
58                 ↪ broken)
59     }
60 }
61
62 // TODO: Return indices range instead of references count
63 public ulong CalculateReferences(ulong link)
64 {
65     var root = GetTreeRoot();
66     var total = GetSize(root);
67     var totalRightIgnore = 0UL;
68     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
152 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
153 {
154     public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)
155         : base(memory)
156     {
157     }
158
159     protected override IntPtr GetLeftPointer(ulong node) => new
160         ↳ IntPtr(&Links[node].LeftAsSource);
161
162     protected override IntPtr GetRightPointer(ulong node) => new
163         ↳ IntPtr(&Links[node].RightAsSource);
164
165     protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsSource;
166     protected override ulong GetRightValue(ulong node) => Links[node].RightAsSource;
167     protected override ulong GetSize(ulong node)
168     {
169         var previousValue = Links[node].SizeAsSource;
170         //return MathHelpers.PartialRead(previousValue, 5, -5);
171         return (previousValue & 4294967264) >> 5;
172     }
173
174     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
175         ↳ left;
176
177     protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
178         ↳ right;
179
180     protected override void SetSize(ulong node, ulong size)
181     {
182         var previousValue = Links[node].SizeAsSource;
183         //var modified = MathHelpers.PartialWrite(previousValue, size, 5, -5);
184         var modified = (previousValue & 31) | ((size & 134217727) << 5);
185         Links[node].SizeAsSource = modified;
186     }
187
188     protected override bool GetLeftIsChild(ulong node)
189     {
190         var previousValue = Links[node].SizeAsSource;
191         //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
192         return (previousValue & 16) >> 4 == 1UL;
193     }
194
195     protected override void SetLeftIsChild(ulong node, bool value)
196     {
197         var previousValue = Links[node].SizeAsSource;
198         //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value, 4,
199         ↳ 1);

```

```

197     var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
198     Links[node].SizeAsSource = modified;
199 }
200
201 protected override bool GetRightIsChild(ulong node)
202 {
203     var previousValue = Links[node].SizeAsSource;
204     //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
205     return (previousValue & 8) >> 3 == 1UL;
206 }
207
208 protected override void SetRightIsChild(ulong node, bool value)
209 {
210     var previousValue = Links[node].SizeAsSource;
211     //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value, 3,
212     ↪ 1);
213     var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
214     Links[node].SizeAsSource = modified;
215 }
216
217 protected override sbyte GetBalance(ulong node)
218 {
219     var previousValue = Links[node].SizeAsSource;
220     //var value = MathHelpers.PartialRead(previousValue, 0, 3);
221     var value = previousValue & 7;
222     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 | 124
223     ↪ : value & 3);
224     return unpackedValue;
225 }
226
227 protected override void SetBalance(ulong node, sbyte value)
228 {
229     var previousValue = Links[node].SizeAsSource;
230     var packagedValue = (ulong)((byte)value >> 5) & 4 | value & 3;
231     //var modified = MathHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
232     var modified = (previousValue & 4294967288) | (packagedValue & 7);
233     Links[node].SizeAsSource = modified;
234 }
235
236 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
237 => Links[first].Source < Links[second].Source ||
238     (Links[first].Source == Links[second].Source && Links[first].Target <
239     ↪ Links[second].Target);
240
241 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
242 => Links[first].Source > Links[second].Source ||
243     (Links[first].Source == Links[second].Source && Links[first].Target >
244     ↪ Links[second].Target);
245
246 protected override ulong GetTreeRoot() => Header->FirstAsSource;
247
248 protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
249
250 /// <summary>
251 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
252 ↪ (концом)
253 /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
254 /// </summary>
255 /// <param name="source">Индекс связи, которая является началом на искомой
256 ↪ связи.</param>
257 /// <param name="target">Индекс связи, которая является концом на искомой
258 ↪ связи.</param>
259 /// <returns>Индекс искомой связи.</returns>
260 public ulong Search(ulong source, ulong target)
261 {
262     var root = Header->FirstAsSource;
263     while (root != 0)
264     {
265         var rootSource = Links[root].Source;
266         var rootTarget = Links[root].Target;
267         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
268         ↪ node.Key < root.Key
269         {
270             root = GetLeftOrDefault(root);
271         }
272         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
273         ↪ node.Key > root.Key
274         {
275             root = GetRightOrDefault(root);
276         }
277         else // node.Key == root.Key
278         {
279             return root;
280         }
281     }
282     return 0;
283 }

```

```

}

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

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

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override void ClearNode(ulong node)
{
    Links[node].LeftAsSource = OUL;
    Links[node].RightAsSource = OUL;
    Links[node].SizeAsSource = OUL;
}

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

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

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override ulong GetTwo() => 2UL;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool ValueEqualToZero(IntPtr pointer) =>
↳ *(ulong*)pointer.ToPointer() == OUL;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

vate class LinksTargetsTreeMethods : LinksTreeMethodsBase

public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
    : base(memory)
{
}
}

```

```

353 //protected override IntPtr GetLeft(ulong node) => new
354     ↳ IntPtr(&Links[node].LeftAsTarget);
355
356 //protected override IntPtr GetRight(ulong node) => new
357     ↳ IntPtr(&Links[node].RightAsTarget);
358
359 //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
360
361 //protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget
362     ↳ = left;
363
364 //protected override void SetRight(ulong node, ulong right) =>
365     ↳ Links[node].RightAsTarget = right;
366
367 //protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget
368     ↳ = size;
369
370 protected override IntPtr GetLeftPointer(ulong node) => new
371     ↳ IntPtr(&Links[node].LeftAsTarget);
372
373 protected override IntPtr GetRightPointer(ulong node) => new
374     ↳ IntPtr(&Links[node].RightAsTarget);
375
376 protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsTarget;
377
378 protected override ulong GetRightValue(ulong node) => Links[node].RightAsTarget;
379
380 protected override ulong GetSize(ulong node)
381 {
382     var previousValue = Links[node].SizeAsTarget;
383     //return MathHelpers.PartialRead(previousValue, 5, -5);
384     return (previousValue & 4294967264) >> 5;
385 }
386
387 protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
388     ↳ left;
389
390 protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget
391     ↳ = right;
392
393 protected override void SetSize(ulong node, ulong size)
394 {
395     var previousValue = Links[node].SizeAsTarget;
396     //var modified = MathHelpers.PartialWrite(previousValue, size, 5, -5);
397     var modified = (previousValue & 31) | ((size & 134217727) << 5);
398     Links[node].SizeAsTarget = modified;
399 }
400
401 protected override bool GetLeftIsChild(ulong node)
402 {
403     var previousValue = Links[node].SizeAsTarget;
404     //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
405     return (previousValue & 16) >> 4 == 1UL;
406     // TODO: Check if this is possible to use
407     //var nodeSize = GetSize(node);
408     //var left = GetLeftValue(node);
409     //var leftSize = GetSizeOrZero(left);
410     //return leftSize > 0 && nodeSize > leftSize;
411 }
412
413 protected override void SetLeftIsChild(ulong node, bool value)
414 {
415     var previousValue = Links[node].SizeAsTarget;
416     //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value, 4,
417     ↳ 1);
418     var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
419     Links[node].SizeAsTarget = modified;
420 }
421
422 protected override bool GetRightIsChild(ulong node)
423 {
424     var previousValue = Links[node].SizeAsTarget;
425     //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
426     return (previousValue & 8) >> 3 == 1UL;
427     // TODO: Check if this is possible to use
428     //var nodeSize = GetSize(node);
429     //var right = GetRightValue(node);
430     //var rightSize = GetSizeOrZero(right);
431     //return rightSize > 0 && nodeSize > rightSize;
432 }
433
434 protected override void SetRightIsChild(ulong node, bool value)
435 {
436     var previousValue = Links[node].SizeAsTarget;
437     //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value, 3,
438     ↳ 1);

```

```

429         var modified = (previousValue & 4294967287) | (((value ? 1UL : 0UL) << 3);
430         Links[node].SizeAsTarget = modified;
431     }
432
433     protected override sbyte GetBalance(ulong node)
434     {
435         var previousValue = Links[node].SizeAsTarget;
436         //var value = MathHelpers.PartialRead(previousValue, 0, 3);
437         var value = previousValue & 7;
438         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 | 124
439             ↪ : value & 3);
440         return unpackedValue;
441     }
442
443     protected override void SetBalance(ulong node, sbyte value)
444     {
445         var previousValue = Links[node].SizeAsTarget;
446         var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
447         //var modified = MathHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
448         var modified = (previousValue & 4294967288) | (packagedValue & 7);
449         Links[node].SizeAsTarget = modified;
450     }
451
452     protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
453     => Links[first].Target < Links[second].Target ||
454         (Links[first].Target == Links[second].Target && Links[first].Source <
455             ↪ Links[second].Source);
456
457     protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
458     => Links[first].Target > Links[second].Target ||
459         (Links[first].Target == Links[second].Target && Links[first].Source >
460             ↪ Links[second].Source);
461
462     protected override ulong GetTreeRoot() => Header->FirstAsTarget;
463
464     protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
465
466     [MethodImpl(MethodImplOptions.AggressiveInlining)]
467     protected override void ClearNode(ulong node)
468     {
469         Links[node].LeftAsTarget = 0UL;
470         Links[node].RightAsTarget = 0UL;
471         Links[node].SizeAsTarget = 0UL;
472     }
473 }
474 }
475 }

```

./Sequences/Converters/BalancedVariantConverter.cs

```

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

```

```

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
15     ///       на этапе сжатия.
16     ///       А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
17     ///       таком случае тип значения элемента массива может быть любым, как char так и ulong.
18     ///       Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
19     ///       пар, а так же разом выполнить замену.
20     /// </remarks>
21     public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
22     {
23         private static readonly LinksCombinedConstants<bool, TLink, long> _constants =
24             ↳ Default<LinksCombinedConstants<bool, TLink, long>>.Instance;
25         private static readonly EqualityComparer<TLink> _equalityComparer =
26             ↳ EqualityComparer<TLink>.Default;
27         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
28
29         private readonly IConverter<IList<TLink>, TLink> _baseConverter;
30         private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
31         private readonly TLink _minFrequencyToCompress;
32         private readonly bool _doInitialFrequenciesIncrement;
33         private Doublet<TLink> _maxDoublet;
34         private LinkFrequency<TLink> _maxDoubletData;
35
36         private struct HalfDoublet
37         {
38             public TLink Element;
39             public LinkFrequency<TLink> DoubletData;
40
41             public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
42             {
43                 Element = element;
44                 DoubletData = doubletData;
45             }
46
47             public override string ToString() => $"{Element}: ({DoubletData})";
48         }
49
50         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
51             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
52             : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
53         {
54         }
55
56         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
57             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
58             ↳ doInitialFrequenciesIncrement)
59             : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
60                 ↳ doInitialFrequenciesIncrement)
61         {
62         }
63
64         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
65             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
66             ↳ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
67             : base(links)
68         {
69             _baseConverter = baseConverter;
70             _doubletFrequenciesCache = doubletFrequenciesCache;
71             if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
72             {
73                 minFrequencyToCompress = Integer<TLink>.One;
74             }
75         }
76     }
77 }

```



```

63     }
64     _minFrequencyToCompress = minFrequencyToCompress;
65     _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
66     ResetMaxDoublet();
67 }
68
69 public override TLink Convert(ICollection<TLink> source) =>
70     ⇨ _baseConverter.Convert(Compress(source));
71
72 /// <remarks>
73 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
74 /// Faster version (doublets' frequencies dictionary is not recreated).
75 /// </remarks>
76 private ICollection<TLink> Compress(ICollection<TLink> sequence)
77 {
78     if (sequence.IsNullOrEmpty())
79     {
80         return null;
81     }
82     if (sequence.Count == 1)
83     {
84         return sequence;
85     }
86     if (sequence.Count == 2)
87     {
88         return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
89     }
90     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
91     var copy = new HalfDoublet[sequence.Count];
92     Doublet<TLink> doublet = default;
93     for (var i = 1; i < sequence.Count; i++)
94     {
95         doublet.Source = sequence[i - 1];
96         doublet.Target = sequence[i];
97         LinkFrequency<TLink> data;
98         if (_doInitialFrequenciesIncrement)
99         {
100             data = _doubletFrequenciesCache.IncrementFrequency(ref 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 increment frequencies,
108                 ⇨ it is expected that all frequencies for the sequence are prepared.");
109             }
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
129 /// <remarks>
130 /// Original algorithm idea: 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, _constants.Null))
141         {
142             _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
143         }
144         var maxDoubletReplacementLink = _maxDoubletData.Link;
145         oldLength--;
146         var oldLengthMinusTwo = oldLength - 1;
147         // Substitute all usages
148         int w = 0, r = 0; // (r == read, w == write)
149         for (; r < oldLength; r++)

```

```

148     {
149         if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
150             ↳ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
151         {
152             if (r > 0)
153             {
154                 var previous = copy[w - 1].Element;
155                 copy[w - 1].DoubletData.DecrementFrequency();
156                 copy[w - 1].DoubletData =
157                     ↳ _doubletFrequenciesCache.IncrementFrequency(previous,
158                     ↳ maxDoubletReplacementLink);
159             }
160             if (r < oldLengthMinusTwo)
161             {
162                 var next = copy[r + 2].Element;
163                 copy[r + 1].DoubletData.DecrementFrequency();
164                 copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxD,
165                     ↳ oubletReplacementLink,
166                     ↳ next);
167             }
168             copy[w++] .Element = maxDoubletReplacementLink;
169             r++;
170             newLength--;
171         }
172         else
173         {
174             copy[w++] = copy[r];
175         }
176         if (w < newLength)
177         {
178             copy[w] = copy[r];
179         }
180         oldLength = newLength;
181         ResetMaxDoublet();
182         UpdateMaxDoublet(copy, newLength);
183     }
184     return newLength;
185 }
186
187 [MethodImpl(MethodImplOptions.AggressiveInlining)]
188 private void ResetMaxDoublet()
189 {
190     _maxDoublet = new Doublet<TLink>();
191     _maxDoubletData = new LinkFrequency<TLink>();
192 }
193
194 [MethodImpl(MethodImplOptions.AggressiveInlining)]
195 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
196 {
197     Doublet<TLink> doublet = default;
198     for (var i = 1; i < length; i++)
199     {
200         doublet.Source = copy[i - 1].Element;
201         doublet.Target = copy[i].Element;
202         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
203     }
204 }
205
206 [MethodImpl(MethodImplOptions.AggressiveInlining)]
207 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
208 {
209     var frequency = data.Frequency;
210     var maxFrequency = _maxDoubletData.Frequency;
211     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
212     ↳ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
213     ↳ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
214     ↳ _maxDoublet.Target)))
215     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
216         (_comparer.Compare(maxFrequency, frequency) < 0 ||
217         ↳ (_equalityComparer.Equals(maxFrequency, frequency) &&
218         ↳ _comparer.Compare(ArithmeticHelpers.Add(doublet.Source, doublet.Target),
219         ↳ ArithmeticHelpers.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
220         ↳ better stability and better compression on sequent data and even on random
221         ↳ numbers data (but gives collisions anyway) */
222     {
223         _maxDoublet = doublet;
224         _maxDoubletData = data;
225     }
226 }

```

./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 =
10             ↳ EqualityComparer<TLink>.Default;
11         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
14
15         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
16             ↳ sequenceToItsLocalElementLevelsConverter) : base(links)
17             ↳ _sequenceToItsLocalElementLevelsConverter =
18                 ↳ sequenceToItsLocalElementLevelsConverter;
19
20         public override TLink Convert(IList<TLink> sequence)
21         {
22             var length = sequence.Count;
23             if (length == 1)
24             {
25                 return sequence[0];
26             }
27             var links = Links;
28             if (length == 2)
29             {
30                 return links.GetOrCreate(sequence[0], sequence[1]);
31             }
32             sequence = sequence.ToArray();
33             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
34             while (length > 2)
35             {
36                 var levelRepeat = 1;
37                 var currentLevel = levels[0];
38                 var previousLevel = levels[0];
39                 var skipOnce = false;
40                 var w = 0;
41                 for (var i = 1; i < length; i++)
42                 {
43                     if (_equalityComparer.Equals(currentLevel, levels[i]))
44                     {
45                         levelRepeat++;
46                         skipOnce = false;
47                         if (levelRepeat == 2)
48                         {
49                             sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
50                             var newLevel = i >= length - 1 ?
51                                 GetPreviousLowerThanCurrentOrCurrent(previousLevel, currentLevel) :
52                                 i < 2 ?
53                                     GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
54                                     GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
55                                         ↳ currentLevel, levels[i + 1]);
56                             levels[w] = newLevel;
57                             previousLevel = currentLevel;
58                             w++;
59                             levelRepeat = 0;
60                             skipOnce = true;
61                         }
62                     }
63                     else if (i == length - 1)
64                     {
65                         {
66                             sequence[w] = sequence[i];
67                             levels[w] = levels[i];
68                             w++;
69                         }
70                     }
71                 }
72             }
73             else
74             {
75                 {
76                     currentLevel = levels[i];
77                     levelRepeat = 1;
78                     if (skipOnce)
79                     {
80                         skipOnce = false;
81                     }
82                 }
83                 else

```

```

74         {
75             sequence[w] = sequence[i - 1];
76             levels[w] = levels[i - 1];
77             previousLevel = levels[w];
78             w++;
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 GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
↪ current, TLink next)
94 {
95     return _comparer.Compare(previous, next) > 0
96         ? _comparer.Compare(previous, current) < 0 ? previous : current
97         : _comparer.Compare(next, current) < 0 ? next : current;
98 }
99
100 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
↪ _comparer.Compare(next, current) < 0 ? next : current;
101
102 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
103 }
104 }

```

./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> : LinksOperatorBase<TLink>,
↪ IConverter<IList<TLink>>
7     {
8         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
9         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
10        public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
↪ IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links) =>
↪ _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
11        public IList<TLink> Convert(IList<TLink> sequence)
12        {
13            var levels = new TLink[sequence.Count];
14            levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
15            for (var i = 1; i < sequence.Count - 1; i++)
16            {
17                var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
18                var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
19                levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
20            }
21            levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
↪ sequence[sequence.Count - 1]);
22            return levels;
23        }
24
25        public TLink GetFrequencyNumber(TLink source, TLink target) =>
↪ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
26    }
27 }

```

./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs

```

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

```

./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 MarkedSequenceCriteriaMatcher<TLink> : ICriteriaMatcher<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 MarkedSequenceCriteriaMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
15         {
16             _links = links;
17             _sequenceMarkerLink = sequenceMarkerLink;
18         }
19
20         public bool IsMatched(TLink sequenceCandidate)
21         => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
22         || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
23             ↳ sequenceCandidate), _links.Constants.Null);
24     }
25 }

```

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

```

./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>, IList<TLink>>>>
10             ↳ _duplicateFragmentsProvider;

```

```

10     public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
        ↳ duplicateFragmentsProvider) => _duplicateFragmentsProvider =
        ↳ duplicateFragmentsProvider;
11     public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
12 }
13 }

```

./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 : IEqualityComparer<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,
        ↳ KeyValuePair<IList<TLink>, IList<TLink>> right) => _listComparer.Equals(left.Key,
        ↳ right.Key) && _listComparer.Equals(left.Value, right.Value);
28             public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
        ↳ HashHelpers.Generate(_listComparer.GetHashCode(pair.Key),
        ↳ _listComparer.GetHashCode(pair.Value));
29         }
30
31         private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
32         {
33             private readonly IListComparer<TLink> _listComparer;
34             public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
35             public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
        ↳ KeyValuePair<IList<TLink>, IList<TLink>> right)
36             {
37                 var intermediateResult = _listComparer.Compare(left.Key, right.Key);
38                 if (intermediateResult == 0)
39                 {
40                     intermediateResult = _listComparer.Compare(left.Value, right.Value);
41                 }
42                 return intermediateResult;
43             }
44         }
45
46         public DuplicateSegmentsProvider(ILinks<TLink> links, ISequences<TLink> sequences)
        : base(minimumStringSegmentLength: 2)
47         {
48             _links = links;
49             _sequences = sequences;
50         }
51
52         public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
53         {
54             _groups = new HashSet<KeyValuePair<IList<TLink>,
        ↳ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
55             var count = _links.Count();
56             _visited = new BitString((long)(Integer<TLink>)count + 1);
57             _links.Each(link =>
58             {
59                 var linkIndex = _links.GetIndex(link);
60                 var linkBitIndex = (long)(Integer<TLink>)linkIndex;
61                 if (!_visited.Get(linkBitIndex))
62                 {
63                     var sequenceElements = new List<TLink>();
64                     _sequences.EachPart(sequenceElements.AddAndReturnTrue, linkIndex);
65                     if (sequenceElements.Count > 2)
66                     {
67                         WalkAll(sequenceElements);
68                     }
69                 }
70             }
71         }

```

```

72     }
73     return _links.Constants.Continue;
74 });
75 var resultList = _groups.ToList();
76 var comparer = Default<ItemComparer>.Instance;
77 resultList.Sort(comparer);
78 #if DEBUG
79     foreach (var item in resultList)
80     {
81         PrintDuplicates(item);
82     }
83 #endif
84     return resultList;
85 }
86
87 protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
↵ length) => new Segment<TLink>(elements, offset, length);
88
89 protected override void OnDuplicateFound(Segment<TLink> segment)
90 {
91     var duplicates = CollectDuplicatesForSegment(segment);
92     if (duplicates.Count > 1)
93     {
94         _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
↵ duplicates));
95     }
96 }
97
98 private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
99 {
100     var duplicates = new List<TLink>();
101     var readAsElement = new HashSet<TLink>();
102     _sequences.Each(sequence =>
103     {
104         duplicates.Add(sequence);
105         readAsElement.Add(sequence);
106         return true; // Continue
107     }, segment);
108     if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
109     {
110         return new List<TLink>();
111     }
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.GetAllPartiallyMatchingSequences4((Has
↵ hSet<ulong>)(object)readAsElement,
↵ (IList<ulong>)segment);
120         foreach (var partiallyMatchedSequence in partiallyMatched)
121         {
122             TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
123             duplicates.Add(sequenceIndex);
124         }
125     }
126     duplicates.Sort();
127     return duplicates;
128 }
129
130 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
131 {
132     if (!_links is ILinks<ulong> ulongLinks)
133     {
134         return;
135     }
136     var duplicatesKey = duplicatesItem.Key;
137     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
138     Console.WriteLine($"{> {keyString} ({string.Join(", ", duplicatesKey))}");
139     var duplicatesList = duplicatesItem.Value;
140     for (int i = 0; i < duplicatesList.Count; i++)
141     {
142         ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
143         var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
↵ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
↵ UnicodeMap.IsCharLink(link.Index) ?
↵ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
144         Console.WriteLine(formattedSequenceStructure);
145         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
↵ ulongLinks);
146         Console.WriteLine(sequenceString);
147     }
148     Console.WriteLine();
149 }

```

```

150     }
151 }

./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> : IIncrementer<IList<TLink>>
7      {
8          private readonly LinkFrequenciesCache<TLink> _cache;
9
10         public FrequenciesCacheBasedLinkFrequencyIncrementer(LinkFrequenciesCache<TLink> cache) =>
11             ↪ _cache = cache;
12
13         /// <remarks>Sequence itseft is not changed, only frequency of its doublets is
14         ↪ incremented.</remarks>
15         public IList<TLink> Increment(IList<TLink> sequence)
16         {
17             _cache.IncrementFrequencies(sequence);
18             return sequence;
19         }
20     }
21 }

```

```

./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToItsFrequencyNumberConverter.cs
1  using Platform.Interfaces;
2
3  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
4  {
5      public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
6          ↪ IConverter<Doublet<TLink>, TLink>
7      {
8          private readonly LinkFrequenciesCache<TLink> _cache;
9          public FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
10             ↪ cache) => _cache = cache;
11         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
12     }
13 }

```

```

./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
11      ↪ between them).
12      /// TODO: Extract interface to implement frequencies storage inside Links storage
13      /// </remarks>
14      public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
15      {
16          private static readonly EqualityComparer<TLink> _equalityComparer =
17             ↪ EqualityComparer<TLink>.Default;
18          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19
20          private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
21          private readonly ICounter<TLink, TLink> _frequencyCounter;
22
23          public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
24             : base(links)
25          {
26              _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
27                 ↪ DoubletComparer<TLink>.Default);
28              _frequencyCounter = frequencyCounter;
29          }
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
33          {
34              var doublet = new Doublet<TLink>(source, target);
35              return GetFrequency(ref doublet);
36          }
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
40          {
41              _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
42              return data;
43          }
44
45          public void IncrementFrequencies(IList<TLink> sequence)
46          {
47
48          }
49      }
50 }

```



```

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,
85                 ↪ _frequencyCounter.Count(link));
86         }
87         _doubletsCache.Add(doublet, data);
88     }
89     return data;
90 }
91
92 public void ValidateFrequencies()
93 {
94     foreach (var entry in _doubletsCache)
95     {
96         var value = entry.Value;
97         var linkIndex = value.Link;
98         if (!_equalityComparer.Equals(linkIndex, default))
99         {
100             var frequency = value.Frequency;
101             var count = _frequencyCounter.Count(linkIndex);
102             // TODO: Why `frequency` always greater than `count` by 1?
103             if (((_comparer.Compare(frequency, count) > 0) &&
104                 ↪ (_comparer.Compare(ArithmeticHelpers.Subtract(frequency, count),
105                 ↪ Integer<TLink>.One) > 0))
106                 || ((_comparer.Compare(count, frequency) > 0) &&
107                 ↪ (_comparer.Compare(ArithmeticHelpers.Subtract(count, frequency),
108                 ↪ Integer<TLink>.One) > 0)))
109             {
110                 throw new InvalidOperationException("Frequencies validation failed.");
111             }
112             //else
113             //{
114             //    if (value.Frequency > 0)
115             //    {
116             //        var frequency = value.Frequency;
117             //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
118             //        var count = _countLinkFrequency(linkIndex);
119             //        if ((frequency > count && frequency - count > 1) || (count > frequency
120             //            ↪ && count - frequency > 1))
121             //            throw new Exception("Frequencies validation failed.");
122             //    }
123             //}

```

./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         public override string ToString() => $"F: {Frequency}, L: {Link}";
28     }
29 }
```

./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, TLink symbol)
12             : base(links, sequenceLink, symbol)
13             => _markedSequenceMatcher = markedSequenceMatcher;
14
15         public override TLink Count()
16         {
17             if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
18             {
19                 return default;
20             }
21             return base.Count();
22         }
23     }
24 }
```

./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 = Comparer<TLink>.Default;
13
14         protected readonly ILinks<TLink> _links;
15         protected readonly TLink _sequenceLink;
16         protected readonly TLink _symbol;
17         protected TLink _total;
18
19         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink, TLink
20             ↪ symbol)
21         {
22             _links = links;
23             _sequenceLink = sequenceLink;
24             _symbol = symbol;
25             _total = default;
26         }
27
28         public virtual TLink Count()
29         {
30             if (_comparer.Compare(_total, default) > 0)
31             {
32                 return _total;
33             }
34             TLink total = default;
35             foreach (TLink link in _links)
36             {
37                 if (link == _symbol)
38                 {
39                     continue;
40                 }
36                 total = _comparer.Compare(link, total) > 0 ? link : total;
37             }
38             return total;
39         }
40     }
41 }
```

```

30         return _total;
31     }
32     StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
33         ↪ IsElement, VisitElement);
34     return _total;
35 }
36
37 private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
38     ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
39     ↪ 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 }
50 }

```

./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs

```

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

```

./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> links,
12             ↪ ICriteriaMatcher<TLink> markedSequenceMatcher, TLink symbol) : base(links, symbol)
13             => _markedSequenceMatcher = markedSequenceMatcher;
14
15         protected override void CountSequenceSymbolFrequency(TLink link)
16         {
17             var symbolFrequencyCounter = new
18                 ↪ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links, _markedSequenceMatcher,
19                 ↪ link, _symbol);
20             _total = ArithmeticHelpers.Add(_total, symbolFrequencyCounter.Count());
21         }
22     }
23 }

```

./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, TLink>
6      {
7          private readonly ILinks<TLink> _links;
8          public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
9          public TLink Count(TLink symbol) => new
10             ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
11     }
12 }

```

./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> : ICounter<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↪ EqualityComparer<TLink>.Default;
11          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13          protected readonly ILinks<TLink> _links;
14          protected readonly TLink _symbol;
15          protected readonly HashSet<TLink> _visits;
16          protected TLink _total;
17
18          public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
19          {
20              _links = links;
21              _symbol = symbol;
22              _visits = new HashSet<TLink>();
23              _total = default;
24          }
25
26          public TLink Count()
27          {
28              if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
29              {
30                  return _total;
31              }
32              CountCore(_symbol);
33              return _total;
34          }
35
36          private void CountCore(TLink link)
37          {
38              var any = _links.Constants.Any;
39              if (_equalityComparer.Equals(_links.Count(any, link), default))
40              {
41                  CountSequenceSymbolFrequency(link);
42              }
43              else
44              {
45                  _links.Each(EachElementHandler, any, link);
46              }
47          }
48
49          protected virtual void CountSequenceSymbolFrequency(TLink link)
50          {
51              var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
52                 ↪ link, _symbol);
53              _total = ArithmeticHelpers.Add(_total, symbolFrequencyCounter.Count());
54          }
55
56          private TLink EachElementHandler(IList<TLink> doublet)
57          {
58              var constants = _links.Constants;
59              var doubletIndex = doublet[constants.IndexPart];
60              if (_visits.Add(doubletIndex))
61              {
62                  CountCore(doubletIndex);
63              }
64              return constants.Continue;
65          }
66      }
67  }

```

./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> : LinksOperatorBase<TLink>,
7             ↪ 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> _propertyOperator;
17
18          public CachedSequenceHeightProvider(
19              ILinks<TLink> links,
20              ISequenceHeightProvider<TLink> baseHeightProvider,
21              IConverter<TLink> addressToUnaryNumberConverter,
22              IConverter<TLink> unaryNumberToAddressConverter,
23              TLink heightPropertyMarker,

```

```

22         IPropertyOperator<TLink, TLink, TLink> propertyOperator)
23         : base(links)
24     {
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, _heightPropertyMarker);
36         if (_equalityComparer.Equals(heightValue, default))
37         {
38             height = _baseHeightProvider.Get(sequence);
39             heightValue = _addressToUnaryNumberConverter.Convert(height);
40             _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
41         }
42         else
43         {
44             height = _unaryNumberToAddressConverter.Convert(heightValue);
45         }
46         return height;
47     }
48 }
49

```

./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> : LinksOperatorBase<TLink>,
7          ↳ ISequenceHeightProvider<TLink>
8      {
9          private readonly ICriteriaMatcher<TLink> _elementMatcher;
10
11         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriteriaMatcher<TLink>
12             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
13
14         public TLink Get(TLink sequence)
15         {
16             var height = default(TLink);
17             var pairOrElement = sequence;
18             while (!_elementMatcher.IsMatched(pairOrElement))
19             {
20                 pairOrElement = Links.GetTarget(pairOrElement);
21                 height = ArithmeticHelpers.Increment(height);
22             }
23             return height;
24         }
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  }
9

```

./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 (последовательностей),
25  /// через естественную группировку по unicode типам, все whitespace вместе, все символы
26  /// ↪ вместе, все числа вместе и т.п.
27  ///
28  /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину графа)
29  ///
30  /// х*у - найти все связи между, в последовательностях любой формы, если не стоит ограничитель
31  /// на то, что является последовательностью, а что нет,
32  /// то находятся любые структуры связей, которые содержат эти элементы именно в таком порядке.
33  ///
34  /// Рост последовательности слева и справа.
35  /// Поиск со звёздочкой.
36  /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
37  /// так же проблема может быть решена при реализации дистанционных триггеров.
38  /// Нужны ли уникальные указатели вообще?
39  /// Что если обращение к информации будет происходить через содержимое всегда?
40  ///
41  /// Писать тесты.
42  ///
43  /// Можно убрать зависимость от конкретной реализации Links,
44  /// на зависимость от абстрактного элемента, который может быть представлен несколькими
45  /// ↪ способами.
46  ///
47  /// Можно ли как-то сделать один общий интерфейс
48  ///
49  /// Блокчейн и/или гит для распределённой записи транзакций.
50  ///
51  /// </remarks>
52  public partial class Sequences : ISequences<ulong> // IList<string>, IList<ulong[]> (после
53  ↪ завершения реализации Sequences)
54  {
55  private static readonly LinksCombinedConstants<bool, ulong, long> _constants =
56  ↪ Default<LinksCombinedConstants<bool, ulong, long>>.Instance;
57
58  /// <summary>Возвращает значение ulong, обозначающее любое количество связей.</summary>
59  public const ulong ZeroOrMany = ulong.MaxValue;
60
61  public SequencesOptions<ulong> Options;
62  public readonly SynchronizedLinks<ulong> Links;
63  public readonly ISynchronization Sync;
64
65  public Sequences(SynchronizedLinks<ulong> links)
66  : this(links, new SequencesOptions<ulong>())
67  {
68  }
69
70  public Sequences(SynchronizedLinks<ulong> links, SequencesOptions<ulong> options)
71  {
72  Links = links;
73  Sync = links.SyncRoot;
74  Options = options;
75
76  Options.ValidateOptions();
77  Options.InitOptions(Links);
78  }
79
80  public bool IsSequence(ulong sequence)
81  {
82  return Sync.ExecuteReadOperation(() =>
83  {
84  if (Options.UseSequenceMarker)
85  {
86  return Options.MarkedSequenceMatcher.IsMatched(sequence);
87  }
88  return !Links.Unsync.IsPartialPoint(sequence);
89  });
90  }
91
92  [MethodImpl(MethodImplOptions.AggressiveInlining)]
93  private ulong GetSequenceByElements(ulong sequence)
94  {
95  if (Options.UseSequenceMarker)
96  {
97  return Links.SearchOrDefault(Options.SequenceMarkerLink, 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  {

```

```

105         return linkContents.Target;
106     }
107     if (linkContents.Target == Options.SequenceMarkerLink)
108     {
109         return linkContents.Source;
110     }
111 }
112 return sequence;
113 }
114
115 #region Count
116
117 public ulong Count(params ulong[] sequence)
118 {
119     if (sequence.Length == 0)
120     {
121         return Links.Count(_constants.Any, Options.SequenceMarkerLink, _constants.Any);
122     }
123     if (sequence.Length == 1) // Первая связь это адрес
124     {
125         if (sequence[0] == _constants.Null)
126         {
127             return 0;
128         }
129         if (sequence[0] == _constants.Any)
130         {
131             return Count();
132         }
133         if (Options.UseSequenceMarker)
134         {
135             return Links.Count(_constants.Any, Options.SequenceMarkerLink, sequence[0]);
136         }
137         return Links.Exists(sequence[0]) ? 1UL : 0;
138     }
139     throw new NotImplementedException();
140 }
141
142 private ulong CountReferences(params ulong[] restrictions)
143 {
144     if (restrictions.Length == 0)
145     {
146         return 0;
147     }
148     if (restrictions.Length == 1) // Первая связь это адрес
149     {
150         if (restrictions[0] == _constants.Null)
151         {
152             return 0;
153         }
154         if (Options.UseSequenceMarker)
155         {
156             var elementsLink = GetSequenceElements(restrictions[0]);
157             var sequenceLink = GetSequenceByElements(elementsLink);
158             if (sequenceLink != _constants.Null)
159             {
160                 return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
161             }
162             return Links.Count(elementsLink);
163         }
164         return Links.Count(restrictions[0]);
165     }
166     throw new NotImplementedException();
167 }
168
169 #endregion
170
171 #region Create
172
173 public ulong Create(params ulong[] sequence)
174 {
175     return Sync.ExecuteWriteOperation(() =>
176     {
177         if (sequence.IsNullOrEmpty())
178         {
179             return _constants.Null;
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 = Options.LinksToSequenceConverter.Convert(sequence);
208     }
209     if (Options.UseSequenceMarker)
210     {
211         Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
212     }
213     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
214 }
215
216 #endregion
217
218 #region Each
219
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, _constants.Any, handler);
242             }
243             return handler(link);
244         }
245         if (sequence.Count == 2)
246         {
247             return Links.Unsync.Each(sequence[0], sequence[1], handler);
248         }
249         if (Options.UseIndex && !Options.Indexer.CheckIndex(sequence))
250         {
251             return false;
252         }
253         return EachCore(handler, sequence);
254     });
255 }
256
257 private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
258 {
259     var matcher = new Matcher(this, sequence, new HashSet<LinkIndex>(), handler);
260     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
261     ↪ Id.
262     Func<ulong, bool> innerHandler = Options.UseSequenceMarker ? (Func<ulong,
263     ↪ bool>)matcher.HandleFullMatchedSequence : matcher.HandleFullMatched;
264     //if (sequence.Length >= 2)
265     if (!StepRight(innerHandler, sequence[0], sequence[1]))
266     {
267         return false;
268     }
269     var last = sequence.Count - 2;
270     for (var i = 1; i < last; i++)
271     {
272         if (!PartialStepRight(innerHandler, sequence[i], sequence[i + 1]))
273         {
274             return false;
275         }
276     }
277     if (sequence.Count >= 3)
278     {

```



```

277         if (!StepLeft(innerHandler, sequence[sequence.Count - 2], sequence[sequence.Count
278             ↪ - 1]))
279         {
280             return false;
281         }
282     return true;
283 }
284
285 private bool PartialStepRight(Func<ulong, bool> handler, ulong left, ulong right)
286 {
287     return Links.Unsync.Each(_constants.Any, left, doublet =>
288     {
289         if (!StepRight(handler, doublet, right))
290         {
291             return false;
292         }
293         if (left != doublet)
294         {
295             return PartialStepRight(handler, doublet, right);
296         }
297         return true;
298     });
299 }
300
301 private bool StepRight(Func<ulong, bool> handler, ulong left, ulong right) =>
302     ↪ Links.Unsync.Each(left, _constants.Any, rightStep => TryStepRightUp(handler, right,
303     ↪ rightStep));
304
305 private bool TryStepRightUp(Func<ulong, bool> handler, ulong right, ulong stepFrom)
306 {
307     var upStep = stepFrom;
308     var firstSource = Links.Unsync.GetTarget(upStep);
309     while (firstSource != right && firstSource != upStep)
310     {
311         upStep = firstSource;
312         firstSource = Links.Unsync.GetSource(upStep);
313     }
314     if (firstSource == right)
315     {
316         return handler(stepFrom);
317     }
318     return true;
319 }
320
321 private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong right) =>
322     ↪ Links.Unsync.Each(_constants.Any, right, leftStep => TryStepLeftUp(handler, left,
323     ↪ leftStep));
324
325 private bool TryStepLeftUp(Func<ulong, bool> handler, ulong left, ulong stepFrom)
326 {
327     var upStep = stepFrom;
328     var firstTarget = Links.Unsync.GetSource(upStep);
329     while (firstTarget != left && firstTarget != upStep)
330     {
331         upStep = firstTarget;
332         firstTarget = Links.Unsync.GetTarget(upStep);
333     }
334     if (firstTarget == left)
335     {
336         return handler(stepFrom);
337     }
338     return true;
339 }
340
341 #endregion
342
343 #region Update
344
345 public ulong Update(ulong[] sequence, ulong[] newSequence)
346 {
347     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
348     {
349         return _constants.Null;
350     }
351     if (sequence.IsNullOrEmpty())
352     {
353         return Create(newSequence);
354     }
355     if (newSequence.IsNullOrEmpty())
356     {
357         Delete(sequence);
358         return _constants.Null;
359     }
360     return Sync.ExecuteWriteOperation(() =>
361     {
362         Links.EnsureEachLinkIsAnyOrExists(sequence);
363         Links.EnsureEachLinkExists(newSequence);
364     });
365 }

```

```

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 последовательности и с
378     // маркером,
379     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
380     // → можно получить имея только фактические последовательности.
381     foreach (var variant in Each(sequence))
382     {
383         if (variant != bestVariant)
384         {
385             UpdateOneCore(variant, bestVariant);
386         }
387     }
388     return bestVariant;
389 }
390
391 private void UpdateOneCore(ulong sequence, ulong newSequence)
392 {
393     if (Options.UseGarbageCollection)
394     {
395         var sequenceElements = GetSequenceElements(sequence);
396         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
397         var sequenceLink = GetSequenceByElements(sequenceElements);
398         var newSequenceElements = GetSequenceElements(newSequence);
399         var newSequenceLink = GetSequenceByElements(newSequenceElements);
400         if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
401         {
402             if (sequenceLink != _constants.Null)
403             {
404                 Links.Unsync.Merge(sequenceLink, newSequenceLink);
405             }
406             Links.Unsync.Merge(sequenceElements, newSequenceElements);
407         }
408         ClearGarbage(sequenceElementsContents.Source);
409         ClearGarbage(sequenceElementsContents.Target);
410     }
411     else
412     {
413         if (Options.UseSequenceMarker)
414         {
415             var sequenceElements = GetSequenceElements(sequence);
416             var sequenceLink = GetSequenceByElements(sequenceElements);
417             var newSequenceElements = GetSequenceElements(newSequence);
418             var newSequenceLink = GetSequenceByElements(newSequenceElements);
419             if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
420             {
421                 if (sequenceLink != _constants.Null)
422                 {
423                     Links.Unsync.Merge(sequenceLink, newSequenceLink);
424                 }
425                 Links.Unsync.Merge(sequenceElements, newSequenceElements);
426             }
427         }
428         else
429         {
430             if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
431             {
432                 Links.Unsync.Merge(sequence, newSequence);
433             }
434         }
435     }
436 }
437
438 #endregion
439
440 #region Delete
441
442 public void Delete(params ulong[] sequence)
443 {
444     Sync.ExecuteWriteOperation(() =>
445     {
446         // TODO: Check all options only ones before loop execution
447     }
448     )
449 }

```

```

444         foreach (var linkToDelete in Each(sequence))
445         {
446             DeleteOneCore(linkToDelete);
447         }
448     });
449 }
450
451 private void DeleteOneCore(ulong link)
452 {
453     if (Options.UseGarbageCollection)
454     {
455         var sequenceElements = GetSequenceElements(link);
456         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
457         var sequenceLink = GetSequenceByElements(sequenceElements);
458         if (Options.UseCascadeDelete || CountReferences(link) == 0)
459         {
460             if (sequenceLink != _constants.Null)
461             {
462                 Links.Unsync.Delete(sequenceLink);
463             }
464             Links.Unsync.Delete(link);
465         }
466         ClearGarbage(sequenceElementsContents.Source);
467         ClearGarbage(sequenceElementsContents.Target);
468     }
469     else
470     {
471         if (Options.UseSequenceMarker)
472         {
473             var sequenceElements = GetSequenceElements(link);
474             var sequenceLink = GetSequenceByElements(sequenceElements);
475             if (Options.UseCascadeDelete || CountReferences(link) == 0)
476             {
477                 if (sequenceLink != _constants.Null)
478                 {
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 возможность,
501 /// гарантировать его использование в других местах).
502 ///
503 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
504 /// </remarks>
505 public ulong Compact(params ulong[] sequence)
506 {
507     return Sync.ExecuteWriteOperation(() =>
508     {
509         if (sequence.IsNullOrEmpty())
510         {
511             return _constants.Null;
512         }
513         Links.EnsureEachLinkExists(sequence);
514         return CompactCore(sequence);
515     });
516 }
517
518 [MethodImpl(MethodImplOptions.AggressiveInlining)]
519 private ulong CompactCore(params ulong[] sequence) => UpdateCore(sequence, sequence);
520
521 #endregion
522
523 #region Garbage Collection
524
525 /// <remarks>
526 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
527 /// ↳ определить извне или в унаследованном классе
528 /// </remarks>
529 [MethodImpl(MethodImplOptions.AggressiveInlining)]
530 private bool IsGarbage(ulong link) => link != Options.SequenceMarkerLink &&
531     ↳ Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;

```

```

530
531 private void ClearGarbage(ulong link)
532 {
533     if (IsGarbage(link))
534     {
535         var contents = new UInt64Link(Links.GetLink(link));
536         Links.Unsync.Delete(link);
537         ClearGarbage(contents.Source);
538         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> patternSequence,
574         ↳ HashSet<LinkIndex> results, Func<LinkIndex, bool> stopableHandler,
575         ↳ HashSet<LinkIndex> readAsElements = null)
576         : base(sequences.Links.Unsync)
577     {
578         _sequences = sequences;
579         _patternSequence = patternSequence;
580         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
581             ↳ _constants.Any && x != ZeroOrMany));
582         _results = results;
583         _stopableHandler = stopableHandler;
584         _readAsElements = readAsElements;
585
586     }
587
588     protected override bool IsElement(IList<ulong> link) => base.IsElement(link) ||
589         ↳ (_readAsElements != null && _readAsElements.Contains(Links.GetIndex(link))) ||
590         ↳ _linksInSequence.Contains(Links.GetIndex(link));
591
592     public bool FullMatch(LinkIndex sequenceToMatch)
593     {
594         _filterPosition = 0;
595         foreach (var part in Walk(sequenceToMatch))
596         {
597             if (!FullMatchCore(Links.GetIndex(part)))
598             {
599                 break;
600             }
601         }
602         return _filterPosition == _patternSequence.Count;
603     }
604
605     private bool FullMatchCore(LinkIndex element)
606     {
607         if (_filterPosition == _patternSequence.Count)
608         {
609             _filterPosition = -2; // Длиннее чем нужно
610             return false;
611         }
612         if (_patternSequence[_filterPosition] != _constants.Any
613             && element != _patternSequence[_filterPosition])
614         {
615             _filterPosition = -1;
616             return false; // Начинается/Продолжается иначе
617         }
618         _filterPosition++;
619         return true;
620     }
621

```

```

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) && _results.Add(sequenceToMatch))
627         {
628             return _stopableHandler(sequenceToMatch);
629         }
630         return true;
631     }
632
633     public bool HandleFullMatchedSequence(ulong sequenceToMatch)
634     {
635         var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
636         if (sequence != _constants.Null && FullMatch(sequenceToMatch) &&
        ↪ _results.Add(sequenceToMatch))
637         {
638             return _stopableHandler(sequence);
639         }
640         return true;
641     }
642
643     /// <remarks>
644     /// TODO: Add support for LinksConstants.Any
645     /// </remarks>
646     public bool PartialMatch(LinkIndex sequenceToMatch)
647     {
648         _filterPosition = -1;
649         foreach (var part in Walk(sequenceToMatch))
650         {
651             if (!PartialMatchCore(Links.GetIndex(part)))
652             {
653                 break;
654             }
655         }
656         return _filterPosition == _patternSequence.Count - 1;
657     }
658
659     private bool PartialMatchCore(LinkIndex element)
660     {
661         if (_filterPosition == (_patternSequence.Count - 1))
662         {
663             return false; // Нашлось
664         }
665         if (_filterPosition >= 0)
666         {
667             if (element == _patternSequence[_filterPosition + 1])
668             {
669                 _filterPosition++;
670             }
671             else
672             {
673                 _filterPosition = -1;
674             }
675         }
676         if (_filterPosition < 0)
677         {
678             if (element == _patternSequence[0])
679             {
680                 _filterPosition = 0;
681             }
682         }
683         return true; // Ищем дальше
684     }
685
686     public void AddPartialMatchedToResults(ulong sequenceToMatch)
687     {
688         if (PartialMatch(sequenceToMatch))
689         {
690             _results.Add(sequenceToMatch);
691         }
692     }
693
694     public bool HandlePartialMatched(ulong sequenceToMatch)
695     {
696         if (PartialMatch(sequenceToMatch))
697         {
698             return _stopableHandler(sequenceToMatch);
699         }

```

```

700         return true;
701     }
702
703     public void AddAllPartialMatchedToResults(IEnumerable<ulong> sequencesToMatch)
704     {
705         foreach (var sequenceToMatch in sequencesToMatch)
706         {
707             if (PartialMatch(sequenceToMatch))
708             {
709                 _results.Add(sequenceToMatch);
710             }
711         }
712     }
713
714     public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<ulong>
715     ↪ sequencesToMatch)
716     {
717         foreach (var sequenceToMatch in sequencesToMatch)
718         {
719             if (PartialMatch(sequenceToMatch))
720             {
721                 _readAsElements.Add(sequenceToMatch);
722                 _results.Add(sequenceToMatch);
723             }
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 sequence.
23         /// </remarks>
24         public ulong[] CreateAllVariants2(ulong[] sequence)
25         {
26             return Sync.ExecuteWriteOperation(() =>
27             {
28                 if (sequence.IsNullOrEmpty())
29                 {
30                     return new ulong[0];
31                 }
32                 Links.EnsureEachLinkExists(sequence);
33                 if (sequence.Length == 1)
34                 {
35                     return sequence;
36                 }
37                 return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
38             });
39         }
40
41         private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
42         {
43             #if DEBUG
44                 if ((stopAt - startAt) < 0)
45                 {
46                     throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть меньше
47                     ↪ или равен stopAt");
48                 }
49                 #endif
50                 if ((stopAt - startAt) == 0)
51                 {
52                     return new[] { sequence[startAt] };
53                 }
54                 if ((stopAt - startAt) == 1)
55                 {

```

```

55     return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt], sequence[stopAt]) };
56 }
57 var variants = new ulong[(ulong)MathHelpers.Catalan(stopAt - startAt)];
58 var last = 0;
59 for (var splitter = startAt; splitter < stopAt; splitter++)
60 {
61     var left = CreateAllVariants2Core(sequence, startAt, splitter);
62     var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
63     for (var i = 0; i < left.Length; i++)
64     {
65         for (var j = 0; j < right.Length; j++)
66         {
67             var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
68             if (variant == _constants.Null)
69             {
70                 throw new NotImplementedException("Creation cancellation is not
71                 ↪ implemented.");
72             }
73             variants[last++] = variant;
74         }
75     }
76     return variants;
77 }
78
79 public List<ulong> CreateAllVariants1(params ulong[] sequence)
80 {
81     return Sync.ExecuteWriteOperation(() =>
82     {
83         if (sequence.IsNullOrEmpty())
84         {
85             return new List<ulong>();
86         }
87         Links.Unsync.EnsureEachLinkExists(sequence);
88         if (sequence.Length == 1)
89         {
90             return new List<ulong> { sequence[0] };
91         }
92         var results = new List<ulong>((int)MathHelpers.Catalan(sequence.Length));
93         return CreateAllVariants1Core(sequence, results);
94     });
95 }
96
97 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
98 {
99     if (sequence.Length == 2)
100     {
101         var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
102         if (link == _constants.Null)
103         {
104             throw new NotImplementedException("Creation cancellation is not implemented.");
105         }
106         results.Add(link);
107         return results;
108     }
109     var innerSequenceLength = sequence.Length - 1;
110     var innerSequence = new ulong[innerSequenceLength];
111     for (var li = 0; li < innerSequenceLength; li++)
112     {
113         var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
114         if (link == _constants.Null)
115         {
116             throw new NotImplementedException("Creation cancellation is not implemented.");
117         }
118         for (var isi = 0; isi < li; isi++)
119         {
120             innerSequence[isi] = sequence[isi];
121         }
122         innerSequence[li] = link;
123         for (var isi = li + 1; isi < innerSequenceLength; isi++)
124         {
125             innerSequence[isi] = sequence[isi + 1];
126         }
127         CreateAllVariants1Core(innerSequence, results);
128     }
129     return results;
130 }
131
132 #endregion
133
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
148 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
149 {
150     if (sequence.Length == 2)
151     {
152         Links.Unsync.Each(sequence[0], sequence[1], handler);
153     }
154     else
155     {
156         var innerSequenceLength = sequence.Length - 1;
157         for (var li = 0; li < innerSequenceLength; li++)
158         {
159             var left = sequence[li];
160             var right = sequence[li + 1];
161             if (left == 0 && right == 0)
162             {
163                 continue;
164             }
165             var linkIndex = li;
166             ulong[] innerSequence = null;
167             Links.Unsync.Each(left, right, doublet =>
168             {
169                 if (innerSequence == null)
170                 {
171                     innerSequence = new ulong[innerSequenceLength];
172                     for (var isi = 0; isi < linkIndex; isi++)
173                     {
174                         innerSequence[isi] = sequence[isi];
175                     }
176                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
177                     {
178                         innerSequence[isi] = sequence[isi + 1];
179                     }
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
203 public void EachPart(Func<ulong, bool> handler, params ulong[] 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
217 private void EachPartCore(Func<ulong, bool> handler, params ulong[] 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, sequence[1]);
388                 if (doublet != _constants.Null)
389                 {
390                     results.Add(doublet);
391                 }
392                 return results;
393             }
394             var linksInSequence = new HashSet<ulong>(sequence);
395             void handler(ulong result)
396             {
397                 var filterPosition = 0;
398                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
399                     ↪ Links.Unsync.GetTarget,
400                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x, x
                     ↪ =>

```

```

401         if (filterPosition == sequence.Length)
402         {
403             filterPosition = -2; // Длиннее чем нужно
404             return false;
405         }
406         if (x != sequence[filterPosition])
407         {
408             filterPosition = -1;
409             return false; // Начинается иначе
410         }
411         filterPosition++;
412
413         return true;
414     });
415     if (filterPosition == sequence.Length)
416     {
417         results.Add(result);
418     }
419 }
420 if (sequence.Length >= 2)
421 {
422     StepRight(handler, sequence[0], sequence[1]);
423 }
424 var last = sequence.Length - 2;
425 for (var i = 1; i < last; i++)
426 {
427     PartialStepRight(handler, sequence[i], sequence[i + 1]);
428 }
429 if (sequence.Length >= 3)
430 {
431     StepLeft(handler, sequence[sequence.Length - 2], sequence[sequence.Length
432         ↪ - 1]);
433 }
434 return results;
435 });
436 }
437
438 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
439 {
440     return Sync.ExecuteReadOperation(() =>
441     {
442         var results = new HashSet<ulong>();
443         if (sequence.Length > 0)
444         {
445             Links.EnsureEachLinkExists(sequence);
446             var firstElement = sequence[0];
447             if (sequence.Length == 1)
448             {
449                 results.Add(firstElement);
450                 return results;
451             }
452             if (sequence.Length == 2)
453             {
454                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
455                 if (doublet != _constants.Null)
456                 {
457                     results.Add(doublet);
458                 }
459                 return results;
460             }
461             var matcher = new Matcher(this, sequence, results, null);
462             if (sequence.Length >= 2)
463             {
464                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
465             }
466             var last = sequence.Length - 2;
467             for (var i = 1; i < last; i++)
468             {
469                 PartialStepRight(matcher.AddFullMatchedToResults, sequence[i], sequence[i
470                     ↪ + 1]);
471             }
472             if (sequence.Length >= 3)
473             {
474                 StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
475                     ↪ sequence[sequence.Length - 1]);
476             }
477             return results;
478         }
479     });
480 }
481
482 public const int MaxSequenceFormatSize = 200;
483
484 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements) =>
485     ↪ FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);

```

```

484 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
    ↳ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↳ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
    ↳ elementToString, insertComma, knownElements));
485
486 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↳ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params LinkIndex[]
    ↳ knownElements)
487 {
488     var linksInSequence = new HashSet<ulong>(knownElements);
489     //var entered = new HashSet<ulong>();
490     var sb = new StringBuilder();
491     sb.Append('{');
492     if (links.Exists(sequenceLink))
493     {
494         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
495             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
    ↳ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
496         {
497             if (insertComma && sb.Length > 1)
498             {
499                 sb.Append(',');
500             }
501             //if (entered.Contains(element))
502             //{
503                 sb.Append('{');
504                 elementToString(sb, element);
505                 sb.Append('}');
506             //}
507             //else
508             elementToString(sb, element);
509             if (sb.Length < MaxSequenceFormatSize)
510             {
511                 return true;
512             }
513             sb.Append(insertComma ? ", ..." : "...");
514             return false;
515         });
516     }
517     sb.Append('}');
518     return sb.ToString();
519 }
520
521 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
    ↳ => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
522
523 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
    ↳ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↳ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
    ↳ sequenceLink, elementToString, insertComma, knownElements));
524
525 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↳ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params LinkIndex[]
    ↳ knownElements)
526 {
527     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,
534             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
    ↳ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
535         {
536             if (insertComma && sb.Length > 1)
537             {
538                 sb.Append(',');
539             }
540             if (entered.Contains(element))
541             {
542                 sb.Append('{');
543                 elementToString(sb, element);
544                 sb.Append('}');
545             }
546             else
547             {
548                 elementToString(sb, element);
549             }
550             if (sb.Length < MaxSequenceFormatSize)
551             {
552                 return true;
553             }
554             sb.Append(insertComma ? ", ..." : "...");
555             return false;

```

```

556         });
557     }
558     sb.Append('}');
559     return sb.ToString();
560 }
561
562 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
563 {
564     return Sync.ExecuteReadOperation(() =>
565     {
566         if (sequence.Length > 0)
567         {
568             Links.EnsureEachLinkExists(sequence);
569             var results = new HashSet<ulong>();
570             for (var i = 0; i < sequence.Length; i++)
571             {
572                 AllUsagesCore(sequence[i], results);
573             }
574             var filteredResults = new List<ulong>();
575             var linksInSequence = new HashSet<ulong>(sequence);
576             foreach (var result in results)
577             {
578                 var filterPosition = -1;
579                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
580                     ↪ Links.Unsync.GetTarget,
581                     ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x, x
582                 {
583                     if (filterPosition == (sequence.Length - 1))
584                     {
585                         return false;
586                     }
587                     if (filterPosition >= 0)
588                     {
589                         if (x == sequence[filterPosition + 1])
590                         {
591                             filterPosition++;
592                         }
593                         else
594                         {
595                             return false;
596                         }
597                     }
598                     if (filterPosition < 0)
599                     {
600                         if (x == sequence[0])
601                         {
602                             filterPosition = 0;
603                         }
604                     }
605                     return true;
606                 });
607                 if (filterPosition == (sequence.Length - 1))
608                 {
609                     filteredResults.Add(result);
610                 }
611             }
612             return filteredResults;
613         }
614         return new List<ulong>();
615     });
616 }
617
618 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
619 {
620     return Sync.ExecuteReadOperation(() =>
621     {
622         if (sequence.Length > 0)
623         {
624             Links.EnsureEachLinkExists(sequence);
625             var results = new HashSet<ulong>();
626             for (var i = 0; i < sequence.Length; i++)
627             {
628                 AllUsagesCore(sequence[i], results);
629             }
630             var filteredResults = new HashSet<ulong>();
631             var matcher = new Matcher(this, sequence, filteredResults, null);
632             matcher.AddAllPartialMatchedToResults(results);
633             return filteredResults;
634         }
635         return new HashSet<ulong>();
636     });
637 }
638
639 public bool GetAllPartiallyMatchingSequences2(Func<ulong, bool> handler, params ulong[]
640     ↪ sequence)
641 {

```

```

640     return Sync.ExecuteReadOperation(() =>
641     {
642         if (sequence.Length > 0)
643         {
644             Links.EnsureEachLinkExists(sequence);
645
646             var results = new HashSet<ulong>();
647             var filteredResults = new HashSet<ulong>();
648             var matcher = new Matcher(this, sequence, filteredResults, handler);
649             for (var i = 0; i < sequence.Length; i++)
650             {
651                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
652                 {
653                     return false;
654                 }
655             }
656             return true;
657         }
658         return true;
659     });
660 }
661
662 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
663 //{
664 //    return Sync.ExecuteReadOperation(() =>
665 //    {
666 //        if (sequence.Length > 0)
667 //        {
668 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
669
670 //            var firstResults = new HashSet<ulong>();
671 //            var lastResults = new HashSet<ulong>();
672
673 //            var first = sequence.First(x => x != LinksConstants.Any);
674 //            var last = sequence.Last(x => x != LinksConstants.Any);
675
676 //            AllUsagesCore(first, firstResults);
677 //            AllUsagesCore(last, lastResults);
678
679 //            firstResults.IntersectWith(lastResults);
680
681 //            //for (var i = 0; i < sequence.Length; i++)
682 //            //    AllUsagesCore(sequence[i], results);
683
684 //            var filteredResults = new HashSet<ulong>();
685 //            var matcher = new Matcher(this, sequence, filteredResults, null);
686 //            matcher.AddAllPartialMatchedToResults(firstResults);
687 //            return filteredResults;
688 //        }
689
690 //        return new HashSet<ulong>();
691 //    });
692 //}
693
694 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
695 {
696     return Sync.ExecuteReadOperation(() =>
697     {
698         if (sequence.Length > 0)
699         {
700             Links.EnsureEachLinkIsAnyOrExists(sequence);
701             var firstResults = new HashSet<ulong>();
702             var lastResults = new HashSet<ulong>();
703             var first = sequence.First(x => x != _constants.Any);
704             var last = sequence.Last(x => x != _constants.Any);
705             AllUsagesCore(first, firstResults);
706             AllUsagesCore(last, lastResults);
707             firstResults.IntersectWith(lastResults);
708             //for (var i = 0; i < sequence.Length; i++)
709             //    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,
720 ↪ IList<ulong> sequence)
721 {
722     return Sync.ExecuteReadOperation(() =>
723     {
724         if (sequence.Count > 0)
725         {
726             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,
755         ↪ readAsElements);
756     matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
757         ↪ x)); // OrderBy is a Hack
758     return filteredResults;
759 }
760 return new HashSet<ulong>();
761 });
762 }
763 // Does not work
764 public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
765     ↪ params ulong[] sequence)
766 {
767     var visited = new HashSet<ulong>();
768     var results = new HashSet<ulong>();
769     var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return true;
770         ↪ }, readAsElements);
771     var last = sequence.Length - 1;
772     for (var i = 0; i < last; i++)
773     {
774         PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
775     }
776     return results;
777 }
778 }
779 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
780 {
781     return Sync.ExecuteReadOperation(() =>
782     {
783         if (sequence.Length > 0)
784         {
785             Links.EnsureEachLinkExists(sequence);
786             //var firstElement = sequence[0];
787             //if (sequence.Length == 1)
788             //{
789                 //results.Add(firstElement);
790                 //return results;
791             //}
792             //if (sequence.Length == 2)
793             //{
794                 //var doublet = _links.SearchCore(firstElement, sequence[1]);
795                 //if (doublet != Doublets.Links.Null)
796                 //    results.Add(doublet);
797                 //return results;
798             //}
799             //var lastElement = sequence[sequence.Length - 1];
800             //Func<ulong, bool> handler = x =>
801             //{
802                 //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
803                 //        ↪ results.Add(x);
804                 //    return true;
805             //};
806             //if (sequence.Length >= 2)
807             //    StepRight(handler, sequence[0], sequence[1]);
808             //var last = sequence.Length - 2;
809             //for (var i = 1; i < last; i++)
810             //    PartialStepRight(handler, sequence[i], sequence[i + 1]);

```

```

807 //if (sequence.Length >= 3)
808 //    StepLeft(handler, sequence[sequence.Length - 2],
809 //        sequence[sequence.Length - 1]);
810 //if (sequence.Length == 1)
811 //if (sequence.Length == 1)
812 //    throw new NotImplementedException(); // all sequences, containing
813 //    this element?
814 //if (sequence.Length == 2)
815 //if (sequence.Length == 2)
816 //    var results = new List<ulong>();
817 //    PartialStepRight(results.Add, sequence[0], sequence[1]);
818 //    return results;
819 //var matches = new List<List<ulong>>();
820 //var last = sequence.Length - 1;
821 //for (var i = 0; i < last; i++)
822 //if (sequence.Length == 1)
823 //    var results = new List<ulong>();
824 //    //StepRight(results.Add, sequence[i], sequence[i + 1]);
825 //    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
826 //    if (results.Count > 0)
827 //        matches.Add(results);
828 //    else
829 //        return results;
830 //    if (matches.Count == 2)
831 //    {
832 //        var merged = new List<ulong>();
833 //        for (var j = 0; j < matches[0].Count; j++)
834 //            for (var k = 0; k < matches[1].Count; k++)
835 //                CloseInnerConnections(merged.Add, matches[0][j],
836 //                    matches[1][k]);
837 //        if (merged.Count > 0)
838 //            matches = new List<List<ulong>> { merged };
839 //        else
840 //            return new List<ulong>();
841 //    }
842 //if (matches.Count > 0)
843 //if (matches.Count > 0)
844 //    var usages = new HashSet<ulong>();
845 //    for (int i = 0; i < sequence.Length; i++)
846 //    {
847 //        AllUsagesCore(sequence[i], usages);
848 //    }
849 //for (int i = 0; i < matches[0].Count; i++)
850 //    // AllUsagesCore(matches[0][i], usages);
851 //usages.UnionWith(matches[0]);
852 //return usages.ToList();
853 //}
854 var firstLinkUsages = new HashSet<ulong>();
855 AllUsagesCore(sequence[0], firstLinkUsages);
856 firstLinkUsages.Add(sequence[0]);
857 //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
858 //    sequence[0] }; // or all sequences, containing this element?
859 //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
860 //    1).ToList();
861 var results = new HashSet<ulong>();
862 foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
863     firstLinkUsages, 1))
864 {
865     AllUsagesCore(match, results);
866 }
867 return results.ToList();
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }

```

/// <remarks>
 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
 /// </remarks>
 public HashSet<ulong> AllUsages(ulong link)
 {
 return Sync.ExecuteReadOperation(() =>
 {
 var usages = new HashSet<ulong>();
 AllUsagesCore(link, usages);
 return usages;
 });
 }
}

// При сборе всех использований (последовательностей) можно сохранять обратный путь к той
 // связи с которой начинался поиск (STTTSSSTT),
 // причём достаточно одного бита для хранения перехода влево или вправо


```

885 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
886 {
887     bool handler(ulong doublet)
888     {
889         if (usages.Add(doublet))
890         {
891             AllUsagesCore(doublet, usages);
892         }
893         return true;
894     }
895     Links.Unsync.Each(link, _constants.Any, handler);
896     Links.Unsync.Each(_constants.Any, link, handler);
897 }
898
899 public HashSet<ulong> AllBottomUsages(ulong link)
900 {
901     return Sync.ExecuteReadOperation(() =>
902     {
903         var visits = new HashSet<ulong>();
904         var usages = new HashSet<ulong>();
905         AllBottomUsagesCore(link, visits, usages);
906         return usages;
907     });
908 }
909
910 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong> usages)
911 {
912     bool handler(ulong doublet)
913     {
914         if (visits.Add(doublet))
915         {
916             AllBottomUsagesCore(doublet, visits, usages);
917         }
918         return true;
919     }
920     if (Links.Unsync.Count(_constants.Any, link) == 0)
921     {
922         usages.Add(link);
923     }
924     else
925     {
926         Links.Unsync.Each(link, _constants.Any, handler);
927         Links.Unsync.Each(_constants.Any, link, handler);
928     }
929 }
930
931 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
932 {
933     if (Options.UseSequenceMarker)
934     {
935         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
936             ↪ Options.MarkedSequenceMatcher, symbol);
937         return counter.Count();
938     }
939     else
940     {
941         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links, symbol);
942         return counter.Count();
943     }
944 }
945
946 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool>
947     ↪ outerHandler)
948 {
949     bool handler(ulong doublet)
950     {
951         if (usages.Add(doublet))
952         {
953             if (!outerHandler(doublet))
954             {
955                 return false;
956             }
957             if (!AllUsagesCore1(doublet, usages, outerHandler))
958             {
959                 return false;
960             }
961         }
962         return true;
963     }
964     return Links.Unsync.Each(link, _constants.Any, handler)
965         && Links.Unsync.Each(_constants.Any, link, handler);
966 }
967
968 public void CalculateAllUsages(ulong[] totals)
969 {
970     var calculator = new AllUsagesCalculator(Links, totals);
971     calculator.Calculate();
972 }

```

```

970 }
971
972 public void CalculateAllUsages2(ulong[] totals)
973 {
974     var calculator = new AllUsagesCalculator2(Links, totals);
975     calculator.Calculate();
976 }
977
978 private class AllUsagesCalculator
979 {
980     private readonly SynchronizedLinks<ulong> _links;
981     private readonly ulong[] _totals;
982
983     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
984     {
985         _links = links;
986         _totals = totals;
987     }
988
989     public void Calculate() => _links.Each(_constants.Any, _constants.Any, CalculateCore);
990
991     private bool CalculateCore(ulong link)
992     {
993         if (_totals[link] == 0)
994         {
995             var total = 1UL;
996             _totals[link] = total;
997             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
1014 private class AllUsagesCalculator2
1015 {
1016     private readonly SynchronizedLinks<ulong> _links;
1017     private readonly ulong[] _totals;
1018
1019     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1020     {
1021         _links = links;
1022         _totals = totals;
1023     }
1024
1025     public void Calculate() => _links.Each(_constants.Any, _constants.Any, CalculateCore);
1026
1027     private bool IsElement(ulong link)
1028     {
1029         // _linksInSequence.Contains(link) ||
1030         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1031             ↪ link;
1032     }
1033
1034     private bool CalculateCore(ulong link)
1035     {
1036         // TODO: Проработать защиту от заикливания
1037         // Основано на SequenceWalker.WalkLeft
1038         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1039         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1040         Func<ulong, bool> isElement = IsElement;
1041         void visitLeaf(ulong parent)
1042         {
1043             if (link != parent)
1044             {
1045                 _totals[parent]++;
1046             }
1047         }
1048         void visitNode(ulong parent)
1049         {
1050             if (link != parent)
1051             {
1052                 _totals[parent]++;
1053             }
1054         }
1055         var stack = new Stack();
1056         var element = link;
1057         if (isElement(element))

```

```

1057     {
1058         visitLeaf(element);
1059     }
1060     else
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     }
1092     _totals[link]++;
1093     return true;
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> usages)
1103     {
1104         _links = links;
1105         _usages = usages;
1106     }
1107
1108     public bool Collect(ulong link)
1109     {
1110         if (_usages.Add(link))
1111         {
1112             _links.Each(link, _constants.Any, Collect);
1113             _links.Each(_constants.Any, link, Collect);
1114         }
1115         return true;
1116     }
1117 }
1118
1119 private class AllUsagesCollector1
1120 {
1121     private readonly ILinks<ulong> _links;
1122     private readonly HashSet<ulong> _usages;
1123     private readonly ulong _continue;
1124
1125     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1126     {
1127         _links = links;
1128         _usages = usages;
1129         _continue = _links.Constants.Continue;
1130     }
1131
1132     public ulong Collect(ICollection<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
1166 private class AllUsagesIntersectingCollector
1167 {
1168     private readonly SynchronizedLinks<ulong> _links;
1169     private readonly HashSet<ulong> _intersectWith;
1170     private readonly HashSet<ulong> _usages;
1171     private readonly HashSet<ulong> _enter;
1172
1173     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
1174     ↪ intersectWith, HashSet<ulong> usages)
1175     {
1176         _links = links;
1177         _intersectWith = intersectWith;
1178         _usages = usages;
1179         _enter = new HashSet<ulong>(); // защита от зацикливания
1180     }
1181
1182     public bool Collect(ulong link)
1183     {
1184         if (_enter.Add(link))
1185         {
1186             if (_intersectWith.Contains(link))
1187             {
1188                 _usages.Add(link);
1189             }
1190             _links.Unsync.Each(link, _constants.Any, Collect);
1191             _links.Unsync.Each(_constants.Any, link, Collect);
1192         }
1193         return true;
1194     }
1195 }
1196
1197 private void CloseInnerConnections(Action<ulong> handler, ulong left, ulong right)
1198 {
1199     TryStepLeftUp(handler, left, right);
1200     TryStepRightUp(handler, right, left);
1201 }
1202
1203 private void AllCloseConnections(Action<ulong> handler, ulong left, ulong right)
1204 {
1205     // Direct
1206     if (left == right)
1207     {
1208         handler(left);
1209     }
1210     var doublet = Links.Unsync.SearchOrDefault(left, right);
1211     if (doublet != _constants.Null)
1212     {
1213         handler(doublet);
1214     }
1215     // Inner
1216     CloseInnerConnections(handler, left, right);
1217     // Outer
1218     StepLeft(handler, left, right);
1219     StepRight(handler, left, right);
1220     PartialStepRight(handler, left, right);
1221     PartialStepLeft(handler, left, right);
1222 }
1223
1224 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
1225 ↪ HashSet<ulong> previousMatchings, long startAt)
1226 {
1227     if (startAt >= sequence.Length) // ?
1228     {
1229         return previousMatchings;
1230     }
1231     var secondLinkUsages = new HashSet<ulong>();
1232     AllUsagesCore(sequence[startAt], secondLinkUsages);
1233     secondLinkUsages.Add(sequence[startAt]);
1234 }

```

```

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,
1238         ↪ secondLinkUsage);
1239         StepRight(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
1240         TryStepRightUp(matchings.AddAndReturnVoid, secondLinkUsage, previousMatching);
1241         //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1242         ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к желаемым
1243         ↪ результатам.
1244         PartialStepRight(matchings.AddAndReturnVoid, previousMatching,
1245         ↪ secondLinkUsage);
1246     }
1247 }
1248 if (matchings.Count == 0)
1249 {
1250     return matchings;
1251 }
1252 return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1253 }
1254
1255 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
1256 ↪ links, params ulong[] sequence)
1257 {
1258     if (sequence == null)
1259     {
1260         return;
1261     }
1262     for (var i = 0; i < sequence.Length; i++)
1263     {
1264         if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany &&
1265         ↪ !links.Exists(sequence[i]))
1266         {
1267             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1268             ↪ $"patternSequence[{i}]");
1269         }
1270     }
1271 }
1272
1273 // Pattern Matching -> Key To Triggers
1274 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1275 {
1276     return Sync.ExecuteReadOperation(() =>
1277     {
1278         patternSequence = Simplify(patternSequence);
1279         if (patternSequence.Length > 0)
1280         {
1281             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1282             var uniqueSequenceElements = new HashSet<ulong>();
1283             for (var i = 0; i < patternSequence.Length; i++)
1284             {
1285                 if (patternSequence[i] != _constants.Any && patternSequence[i] !=
1286                 ↪ ZeroOrMany)
1287                 {
1288                     uniqueSequenceElements.Add(patternSequence[i]);
1289                 }
1290             }
1291             var results = new HashSet<ulong>();
1292             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1293             {
1294                 AllUsagesCore(uniqueSequenceElement, results);
1295             }
1296             var filteredResults = new HashSet<ulong>();
1297             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1298             matcher.AddAllPatternMatchedToResults(results);
1299             return filteredResults;
1300         }
1301         return new HashSet<ulong>();
1302     });
1303 }
1304
1305 // Найти все возможные связи между указанным списком связей.
1306 // Находит связи между всеми указанными связями в любом порядке.
1307 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются несколько
1308 ↪ раз в последовательности)
1309 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1310 {
1311     return Sync.ExecuteReadOperation(() =>
1312     {
1313         var results = new HashSet<ulong>();
1314         if (linksToConnect.Length > 0)
1315         {
1316             Links.EnsureEachLinkExists(linksToConnect);
1317         }
1318     });
1319 }

```

```

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
1320 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1321 {
1322     return Sync.ExecuteReadOperation(() =>
1323     {
1324         var results = new HashSet<ulong>();
1325         if (linksToConnect.Length > 0)
1326         {
1327             Links.EnsureEachLinkExists(linksToConnect);
1328             var collector1 = new AllUsagesCollector(Links.Unsync, 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, next);
1334                 collector.Collect(linksToConnect[i]);
1335                 results.IntersectWith(next);
1336                 next.Clear();
1337             }
1338         }
1339         return results;
1340     });
1341 }
1342
1343 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1344 {
1345     return Sync.ExecuteReadOperation(() =>
1346     {
1347         var results = new HashSet<ulong>();
1348         if (linksToConnect.Length > 0)
1349         {
1350             Links.EnsureEachLinkExists(linksToConnect);
1351             var collector1 = new AllUsagesCollector(Links, results);
1352             collector1.Collect(linksToConnect[0]);
1353             //AllUsagesCore(linksToConnect[0], results);
1354             for (var i = 1; i < linksToConnect.Length; i++)
1355             {
1356                 var next = new HashSet<ulong>();
1357                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1358                 collector.Collect(linksToConnect[i]);
1359                 //AllUsagesCore(linksToConnect[i], next);
1360                 //results.IntersectWith(next);
1361                 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); // new
1373         ⇨ BitArray((int)_links.Total + 1);
1374         if (linksToConnect.Length > 0)
1375         {
1376             Links.EnsureEachLinkExists(linksToConnect);
1377             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1378             collector1.Collect(linksToConnect[0]);
1379             for (var i = 1; i < linksToConnect.Length; i++)
1380             {
1381                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1382                 ⇨ BitArray((int)_links.Total + 1);
1383                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1384                 collector.Collect(linksToConnect[i]);
1385                 results = results.And(next);
1386             }
1387         }
1388         return results.GetSetUInt64Indices();
1389     });
1390 }
1391
1392 private static ulong[] Simplify(ulong[] sequence)
1393 {

```

```

1392 // Считаем новый размер последовательности
1393 long newLength = 0;
1394 var zeroOrManyStepped = false;
1395 for (var i = 0; i < sequence.Length; i++)
1396 {
1397     if (sequence[i] == ZeroOrMany)
1398     {
1399         if (zeroOrManyStepped)
1400         {
1401             continue;
1402         }
1403         zeroOrManyStepped = true;
1404     }
1405     else
1406     {
1407         //if (zeroOrManyStepped) Is it efficient?
1408         zeroOrManyStepped = false;
1409     }
1410     newLength++;
1411 }
1412 // Строим новую последовательность
1413 zeroOrManyStepped = false;
1414 var newSequence = new ulong[newLength];
1415 long j = 0;
1416 for (var i = 0; i < sequence.Length; i++)
1417 {
1418     //var current = zeroOrManyStepped;
1419     //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1420     //if (current && zeroOrManyStepped)
1421     //    continue;
1422     //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1423     //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,
1447     ↪ ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1448     var simplifiedSequence = Simplify(sequence);
1449 }
1450
1451 public List<ulong> GetSimilarSequences() => new List<ulong>();
1452
1453 public void Prediction()
1454 {
1455     //_links
1456     //_sequences
1457 }
1458
1459 #region From Triplets
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,
1480     ↳ ulong rightLink, int rightBound, ref List<ulong> results)
1481 {
1482     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1483     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1484     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1485     {
1486         var nextLeftLink = middleLinks[leftBound];
1487         var elements = GetRightElements(leftLink, nextLeftLink);
1488         if (leftBound <= rightBound)
1489         {
1490             for (var i = elements.Length - 1; i >= 0; i--)
1491             {
1492                 var element = elements[i];
1493                 if (element != 0)
1494                 {
1495                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1496                         ↳ rightLink, rightBound, ref results);
1497                 }
1498             }
1499         }
1500         else
1501         {
1502             for (var i = elements.Length - 1; i >= 0; i--)
1503             {
1504                 var element = elements[i];
1505                 if (element != 0)
1506                 {
1507                     results.Add(element);
1508                 }
1509             }
1510         }
1511     }
1512     else
1513     {
1514         var nextRightLink = middleLinks[rightBound];
1515         var elements = GetLeftElements(rightLink, nextRightLink);
1516         if (leftBound <= rightBound)
1517         {
1518             for (var i = elements.Length - 1; i >= 0; i--)
1519             {
1520                 var element = elements[i];
1521                 if (element != 0)
1522                 {
1523                     CollectMatchingSequences(leftLink, leftBound, middleLinks,
1524                         ↳ elements[i], rightBound - 1, ref results);
1525                 }
1526             }
1527         }
1528         else
1529         {
1530             for (var i = elements.Length - 1; i >= 0; i--)
1531             {
1532                 var element = elements[i];
1533                 if (element != 0)
1534                 {
1535                     results.Add(element);
1536                 }
1537             }
1538         }
1539     }
1540 }
1541
1542 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1543 {
1544     var result = new ulong[5];
1545     TryStepRight(startLink, rightLink, result, 0);
1546     Links.Each(_constants.Any, startLink, couple =>
1547     {
1548         if (couple != startLink)
1549         {
1550             if (TryStepRight(couple, rightLink, result, 2))
1551             {
1552                 return false;
1553             }
1554         }
1555         return true;
1556     });
1557     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1558     {
1559         result[4] = startLink;
1560     }
1561     return result;
1562 }
1563
1564 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 ==
1578                 ↪ Net.And &&
1579             {
1580                 result[offset + 1] = couple;
1581                 if (++added == 2)
1582                 {
1583                     return false;
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             return true;
1604         });
1605     if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1606     {
1607         result[4] = leftLink;
1608     }
1609     return result;
1610 }
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 ==
1630                 ↪ Net.And &&
1631             {
1632                 result[offset + 1] = couple;
1633                 if (++added == 2)
1634                 {
1635                     return false;
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
1676     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1677         ↳ HashSet<LinkIndex> results)
1678         : base(sequences.Links.Unsync)
1679     {
1680         _sequences = sequences;
1681         _patternSequence = patternSequence;
1682         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1683             ↳ _constants.Any && x != ZeroOrMany));
1684         _results = results;
1685         _pattern = CreateDetailedPattern();
1686
1687     }
1688
1689     protected override bool IsElement(IList<ulong> link) =>
1690         ↳ _linksInSequence.Contains(Links.GetIndex(link)) || base.IsElement(link);
1691
1692     public bool PatternMatch(LinkIndex sequenceToMatch)
1693     {
1694         _patternPosition = 0;
1695         _sequencePosition = 0;
1696         foreach (var part in Walk(sequenceToMatch))
1697         {
1698             if (!PatternMatchCore(Links.GetIndex(part)))
1699             {
1700                 break;
1701             }
1702         }
1703         return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count -
1704             ↳ 1 && _pattern[_patternPosition].Start == 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

```

```

1733         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 /** matchhere: search for regexp at beginning of text */
1799 /**int matchhere(char* regexp, char* text)
1800 /**{
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 /**    {
1817 /**        /* a * matches zero or more instances */
1818 /**        if (matchhere(regexp, text))
1819 /**            return 1;
1820 /**    } while (*text != '\0' && (*text++ == c || c == '.'));
1821 /**    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         ///////
1914         //if (_filterPosition == _patternSequence.Length)
1915         //{
1916         //      _filterPosition = -2; // Длиннее чем нужно
1917         //      return false;
1918         //}
1919         //if (element != _patternSequence[_filterPosition])
1920         //{
1921         //      _filterPosition = -1;
1922         //      return false; // Начинается иначе
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
1953 #endregion
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             hasElements = false;
34             for (var i = 0; i < array.Length; i++)
35             {
36                 var candidate = array[i];

```

```

37         if (candidate == 0)
38         {
39             continue;
40         }
41         var doubletOffset = i * 2;
42         if (isElement(candidate))
43         {
44             nextArray[doubletOffset] = candidate;
45         }
46         else
47         {
48             var link = links.GetLink(candidate);
49             var linkSource = links.GetSource(link);
50             var linkTarget = links.GetTarget(link);
51             nextArray[doubletOffset] = linkSource;
52             nextArray[doubletOffset + 1] = linkTarget;
53             if (!hasElements)
54             {
55                 hasElements = !(isElement(linkSource) && isElement(linkTarget));
56             }
57         }
58     }
59     #if USEARRAYPOOL
60     if (array.Length > 1)
61     {
62         ArrayPool.Free(array);
63     }
64 #endif
65     array = nextArray;
66 }
67 while (hasElements);
68 var filledElementsCount = CountFilledElements(array);
69 if (filledElementsCount == array.Length)
70 {
71     return array;
72 }
73 else
74 {
75     return CopyFilledElements(array, filledElementsCount);
76 }
77 }
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 private static ulong[] CopyFilledElements(ulong[] array, int filledElementsCount)
81 {
82     var finalArray = new ulong[filledElementsCount];
83     for (int i = 0, j = 0; i < array.Length; i++)
84     {
85         if (array[i] > 0)
86         {
87             finalArray[j] = array[i];
88             j++;
89         }
90     }
91     #if USEARRAYPOOL
92     ArrayPool.Free(array);
93 #endif
94     return finalArray;
95 }
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 private static int CountFilledElements(ulong[] array)
99 {
100     var count = 0;
101     for (var i = 0; i < array.Length; i++)
102     {
103         if (array[i] > 0)
104         {
105             count++;
106         }
107     }
108     return count;
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, IList<TLink[]>
9             ↪ groupedSequence)
10         {
11             var finalSequence = new TLink[groupedSequence.Count];

```

```

11         for (var i = 0; i < finalSequence.Length; i++)
12         {
13             var part = groupedSequence[i];
14             finalSequence[i] = part.Length == 1 ? part[0] : sequences.Create(part);
15         }
16         return sequences.Create(finalSequence);
17     }
18 }
19 }

```

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

```

```

71         return indexed;
72     }
73
74     public bool CheckIndex(IList<TLink> sequence)
75     {
76         var indexed = true;
77         var i = sequence.Count;
78         while (--i >= 1 && (indexed =
79             ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
80             ↪ _null))) { }
81         return indexed;
82     }
83 }

```

./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 <TLink> the ILinks<TLink>
12     ↪ 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 update/delete.
21         public bool UseSequenceMarker { get; set; }
22         public bool UseCompression { get; set; }
23         public bool UseGarbageCollection { get; set; }
24         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
25         public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
26
27         public MarkedSequenceCreteriaMatcher<TLink> MarkedSequenceMatcher { get; set; }
28         public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
29         public SequencesIndexer<TLink> Indexer { get; set; }
30
31         // TODO: Реализовать компактификацию при чтении
32         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
33         //public bool UseRequestMarker { get; set; }
34         //public bool StoreRequestResults { get; set; }
35
36         public void InitOptions(ISynchronizedLinks<TLink> links)
37         {
38             if (UseSequenceMarker)
39             {
40                 if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
41                 {
42                     SequenceMarkerLink = links.CreatePoint();
43                 }
44                 else
45                 {
46                     if (!links.Exists(SequenceMarkerLink))
47                     {
48                         var link = links.CreatePoint();
49                         if (!_equalityComparer.Equals(link, SequenceMarkerLink))
50                         {
51                             throw new InvalidOperationException("Cannot recreate sequence marker
52                             ↪ link.");
53                         }
54                     }
55                 }
56             }
57             if (MarkedSequenceMatcher == null)
58             {
59                 MarkedSequenceMatcher = new MarkedSequenceCreteriaMatcher<TLink>(links,
60                 ↪ SequenceMarkerLink);
61             }
62             var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
63             if (UseCompression)
64             {
65                 if (LinksToSequenceConverter == null)
66                 {
67                     ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
68                     if (UseSequenceMarker)
69                     {
70                         totalSequenceSymbolFrequencyCounter = new
71                         ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
72                         ↪ MarkedSequenceMatcher);

```



```

67     }
68     else
69     {
70         totalSequenceSymbolFrequencyCounter = new
            ↳ TotalSequenceSymbolFrequencyCounter<TLink>(links);
71     }
72     var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
            ↳ totalSequenceSymbolFrequencyCounter);
73     var compressingConverter = new CompressingConverter<TLink>(links,
            ↳ balancedVariantConverter, doubletFrequenciesCache);
74     LinksToSequenceConverter = compressingConverter;
75 }
76 }
77 else
78 {
79     if (LinksToSequenceConverter == null)
80     {
81         LinksToSequenceConverter = balancedVariantConverter;
82     }
83 }
84 if (UseIndex && Indexer == null)
85 {
86     Indexer = new SequencesIndexer<TLink>(links);
87 }
88 }
89
90 public void ValidateOptions()
91 {
92     if (UseGarbageCollection && !UseSequenceMarker)
93     {
94         throw new NotSupportedException("To use garbage collection UseSequenceMarker
            ↳ option must be on.");
95     }
96 }
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 + char.MaxValue;
14         public static readonly ulong MapSize = 1 + char.MaxValue;
15
16         private readonly ILinks<ulong> _links;
17         private bool _initialized;
18
19         public UnicodeMap(ILinks<ulong> links) => _links = links;
20
21         public static UnicodeMap InitNew(ILinks<ulong> links)
22         {
23             var map = new UnicodeMap(links);
24             map.Init();
25             return map;
26         }
27
28         public void Init()
29         {
30             if (_initialized)
31             {
32                 return;
33             }
34             _initialized = true;
35             var firstLink = _links.CreatePoint();
36             if (firstLink != FirstCharLink)
37             {
38                 _links.Delete(firstLink);
39             }
40             else
41             {
42                 for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
43                 {
44                     // From NIL to It (NIL -> Character) transformation meaning, (or infinite
                        ↳ amount of NIL characters before actual Character)
45                     var createdLink = _links.CreatePoint();
46                     _links.Update(createdLink, firstLink, createdLink);
47                     if (createdLink != i)
48                     {
49                         throw new InvalidOperationException("Unable to initialize UTF 16 table.");

```

```

50     }
51 }
52 }
53 }
54
55 // 0 - null link
56 // 1 - nil character (0 character)
57 // ...
58 // 65536 (0(1) + 65535 = 65536 possible values)
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public static ulong FromCharToLink(char character) => (ulong)character + 1;
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 public static char FromLinkToChar(ulong link) => (char)(link - 1);
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 public static bool IsCharLink(ulong link) => link <= MapSize;
68
69 public static string FromLinksToString(IList<ulong> linksList)
70 {
71     var sb = new StringBuilder();
72     for (int i = 0; i < linksList.Count; i++)
73     {
74         sb.Append(FromLinkToChar(linksList[i]));
75     }
76     return sb.ToString();
77 }
78
79 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
80 {
81     var sb = new StringBuilder();
82     if (links.Exists(link))
83     {
84         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
85             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
86             ↪ element =>
87             {
88                 sb.Append(FromLinkToChar(element));
89                 return true;
90             }
91         );
92     }
93     return sb.ToString();
94 }
95
96 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
97     ↪ chars.Length);
98
99 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
100 {
101     // char array to ulong array
102     var linksSequence = new ulong[count];
103     for (var i = 0; i < count; i++)
104     {
105         linksSequence[i] = FromCharToLink(chars[i]);
106     }
107     return linksSequence;
108 }
109
110 public static ulong[] FromStringToLinkArray(string sequence)
111 {
112     // char array to ulong array
113     var linksSequence = new ulong[sequence.Length];
114     for (var i = 0; i < sequence.Length; i++)
115     {
116         linksSequence[i] = FromCharToLink(sequence[i]);
117     }
118     return linksSequence;
119 }
120
121 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
122 {
123     var result = new List<ulong[]>();
124     var offset = 0;
125     while (offset < sequence.Length)
126     {
127         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
128         var relativeLength = 1;
129         var absoluteLength = offset + relativeLength;
130         while (absoluteLength < sequence.Length &&
131             currentCategory ==
132             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
133         {
134             relativeLength++;
135             absoluteLength++;
136         }
137         // 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 =
156                 ↪ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
157             var absoluteLength = offset + relativeLength;
158             while (absoluteLength < array.Length &&
159                 array[absoluteLength] <= LastCharLink &&
160                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(ar
161                 ↪ ray[absoluteLength])))
162             {
163                 relativeLength++;
164                 absoluteLength++;
165             }
166         }
167         else
168         {
169             var absoluteLength = offset + relativeLength;
170             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
171             {
172                 relativeLength++;
173                 absoluteLength++;
174             }
175             // copy array
176             var innerSequence = new ulong[relativeLength];
177             var maxLength = offset + relativeLength;
178             for (var i = offset; i < maxLength; i++)
179             {
180                 innerSequence[i - offset] = array[i];
181             }
182             result.Add(innerSequence);
183             offset += relativeLength;
184         }
185     }
186     return result;
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) =>
12             ↪ Links.GetLink(Links.GetSource(element));
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override IList<TLink> GetNextElementAfterPush(IList<TLink> element) =>
16             ↪ Links.GetLink(Links.GetTarget(element));
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override IEnumerable<IList<TLink>> WalkContents(IList<TLink> element)
20         {
21             var start = Links.Constants.IndexPart + 1;
22             for (var i = element.Count - 1; i >= start; i--)
23             {
24                 var partLink = Links.GetLink(element[i]);
25                 if (IsElement(partLink))
26                 {
27                     yield return partLink;
28                 }
29             }
30         }
31     }
32 }

```

```

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) =>
12             Links.GetLink(Links.GetTarget(element));
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override IList<TLink> GetNextElementAfterPush(IList<TLink> element) =>
16             Links.GetLink(Links.GetSource(element));
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override IEnumerable<IList<TLink>> WalkContents(IList<TLink> element)
20         {
21             for (var i = Links.Constants.IndexPart + 1; i < element.Count; i++)
22             {
23                 var partLink = Links.GetLink(element[i]);
24                 if (IsElement(partLink))
25                 {
26                     yield return partLink;
27                 }
28             }
29         }
30     }
31 }

```

./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> : LinksOperatorBase<TLink>,
8          ↳ ISequenceWalker<TLink>
9      {
10         // TODO: Use IStack inead of System.Collections.Generic.Stack, but IStack should contain
11         ↳ IsEmpty property
12         private readonly Stack<IList<TLink>> _stack;
13
14         protected SequenceWalkerBase(ILinks<TLink> links) : base(links) => _stack = new
15             ↳ 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 while(!_stack.IsEmpty) _stack.Pop()
22             }
23             var element = Links.GetLink(sequence);
24             if (IsElement(element))
25             {
26                 yield return element;
27             }
28             else
29             {
30                 while (true)
31                 {
32                     if (IsElement(element))
33                     {
34                         if (_stack.Count == 0)
35                         {
36                             break;
37                         }
38                         element = _stack.Pop();
39                         foreach (var output in WalkContents(element))
40                         {
41                             yield return output;
42                         }
43                         element = GetNextElementAfterPop(element);
44                     }
45                     else
46                     {
47                         _stack.Push(element);
48                         element = GetNextElementAfterPush(element);
49                     }
50                 }
51             }
52         }
53     }
54 }

```

```

48     }
49 }
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected virtual bool IsElement(IList<TLink> elementLink) =>
53     ⇨ Point<TLink>.IsPartialPointUnchecked(elementLink);
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 protected abstract IList<TLink> GetNextElementAfterPop(IList<TLink> element);
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 protected abstract IList<TLink> GetNextElementAfterPush(IList<TLink> element);
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected abstract IEnumerable<IList<TLink>> WalkContents(IList<TLink> element);
63 }

```

./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, GetStackMarker(),
40              ⇨ _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 stackMarker)
6          {
7              var stackPoint = links.CreatePoint();
8              var stack = links.Update(stackPoint, stackMarker, stackPoint);
9              return stack;
10          }
11
12          public static void DeleteStack<TLink>(this ILinks<TLink> links, TLink stack) =>
13              ⇨ links.Delete(stack);
14      }
15 }

```

./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 performance improvements.
12     /// TODO: Or even to unfold multiple layers of implementations.
13     /// </remarks>
14     public class SynchronizedLinks<T> : ISynchronizedLinks<T>
15     {
16         public LinksCombinedConstants<T, T, int> Constants { get; }
17         public ISynchronization SyncRoot { get; }
18         public ILinks<T> Sync { get; }
19         public ILinks<T> Unsync { get; }
20
21         public SynchronizedLinks(ILinks<T> links) : this(new ReaderWriterLockSynchronization(),
22             ↪ links) { }
23
24         public SynchronizedLinks(ISynchronization synchronization, ILinks<T> links)
25         {
26             SyncRoot = synchronization;
27             Sync = this;
28             Unsync = links;
29             Constants = links.Constants;
30         }
31
32         public T Count(IList<T> restriction) => SyncRoot.ExecuteReadOperation(restriction,
33             ↪ Unsync.Count);
34         public T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
35             ↪ SyncRoot.ExecuteReadOperation(handler, restrictions, (handler1, restrictions1) =>
36             ↪ Unsync.Each(handler1, restrictions1));
37         public T Create() => SyncRoot.ExecuteWriteOperation(Unsync.Create);
38         public T Update(IList<T> restrictions) => SyncRoot.ExecuteWriteOperation(restrictions,
39             ↪ Unsync.Update);
40         public void Delete(T link) => SyncRoot.ExecuteWriteOperation(link, Unsync.Delete);
41
42         //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
43         //    ↪ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
44         //{
45         //    if (restriction != null && substitution != null &&
46         //        ↪ !substitution.EqualTo(restriction))
47         //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler, substitution,
48         //            ↪ substitutedHandler, Unsync.Trigger);
49         //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
50         //        ↪ substitutedHandler, Unsync.Trigger);
51         //}
52     }
53 }

```

./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>, IList<ulong>
15     {
16         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
17             ↪ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
18
19         private const int Length = 3;
20
21         public readonly ulong Index;
22         public readonly ulong Source;
23         public readonly ulong Target;
24
25         public static readonly UInt64Link Null = new UInt64Link();
26
27         public UInt64Link(params ulong[] values)
28         {
29             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
30                 ↪ _constants.Null;
31             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
32                 ↪ _constants.Null;
33             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
34                 ↪ _constants.Null;
35         }
36
37         public UInt64Link(IList<ulong> values)
38         {
39             Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
40                 ↪ _constants.Null;

```

```

36         Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
37         ↪ _constants.Null;
38     Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
39     ↪ _constants.Null;
40 }
41
42 public UInt64Link(ulong index, ulong source, ulong target)
43 {
44     Index = index;
45     Source = source;
46     Target = target;
47 }
48
49 public UInt64Link(ulong source, ulong target)
50 : this(_constants.Null, source, target)
51 {
52     Source = source;
53     Target = target;
54 }
55
56 public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
57 ↪ target);
58
59 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
60
61 public bool IsNull() => Index == _constants.Null
62     && Source == _constants.Null
63     && Target == _constants.Null;
64
65 public override bool Equals(object other) => other is UInt64Link &&
66 ↪ Equals((UInt64Link)other);
67
68 public bool Equals(UInt64Link other) => Index == other.Index
69     && Source == other.Source
70     && Target == other.Target;
71
72 public static string ToString(ulong index, ulong source, ulong target) => $"{index}:
73 ↪ {source}->{target}";
74
75 public static string ToString(ulong source, ulong target) => $"{source}->{target}";
76
77 public static implicit operator ulong[] (UInt64Link link) => link.ToArray();
78
79 public static implicit operator UInt64Link(ulong[] linkArray) => new 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 ? ToString(Source, Target) :
89 ↪ 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 - 1), nameof(index));
98         if (index == _constants.IndexPart)
99         {
100             return Index;
101         }
102         if (index == _constants.SourcePart)
103         {
104             return Source;
105         }
106         if (index == _constants.TargetPart)
107         {
108             return Target;
109         }
110         throw new NotSupportedException(); // Impossible path due to Ensure.ArgumentInRange
111     }
112     set => throw new NotSupportedException();
113 }
114
115 public int Count => Length;
116
117 public bool IsReadOnly => true;
118
119 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
120
121 public IEnumerator<ulong> GetEnumerator()
122 {
123     yield return Index;
124     yield return Source;
125 }

```

```

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

```

./UInt64LinkExtensions.cs

```

1 namespace Platform.Data.Doublets
2 {
3     public static class UInt64LinkExtensions
4     {
5         public static bool IsFullPoint(this UInt64Link link) => Point<ulong>.IsFullPoint(link);
6         public static bool IsPartialPoint(this UInt64Link link) =>
7             ↪ Point<ulong>.IsPartialPoint(link);
8     }
9 }

```

./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> Constants =
14             ↪ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
15
16         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
17
18         public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
19         {
20             if (sequence == null)
21             {
22                 return;
23             }
24             for (var i = 0; i < sequence.Count; i++)
25             {

```



```

25         if (!links.Exists(sequence[i]))
26         {
27             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
28                 ↳ $"sequence[{i}]");
29         }
30     }
31 }
32 public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
33     ↳ sequence)
34 {
35     if (sequence == null)
36     {
37         return;
38     }
39     for (var i = 0; i < sequence.Count; i++)
40     {
41         if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
42         {
43             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
44                 ↳ $"sequence[{i}]");
45         }
46     }
47 }
48 public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
49 {
50     if (sequence == null)
51     {
52         return false;
53     }
54     var constants = links.Constants;
55     for (var i = 0; i < sequence.Length; i++)
56     {
57         if (sequence[i] == constants.Any)
58         {
59             return true;
60         }
61     }
62     return false;
63 }
64 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
65     ↳ Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
66 {
67     var sb = new StringBuilder();
68     var visited = new HashSet<ulong>();
69     links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
70         ↳ innerSb.Append(link.Index), renderIndex, renderDebug);
71     return sb.ToString();
72 }
73 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
74     ↳ Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
75     ↳ bool renderIndex = false, bool renderDebug = false)
76 {
77     var sb = new StringBuilder();
78     var visited = new HashSet<ulong>();
79     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
80         ↳ renderDebug);
81     return sb.ToString();
82 }
83 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
84     ↳ HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
85     ↳ Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false, bool
86     ↳ renderDebug = false)
87 {
88     if (sb == null)
89     {
90         throw new ArgumentNullException(nameof(sb));
91     }
92     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
93         ↳ Constants.Itself)
94     {
95         return;
96     }
97     if (links.Exists(linkIndex))
98     {
99         if (visited.Add(linkIndex))
100         {
101             sb.Append('(');
102             var link = new UInt64Link(links.GetLink(linkIndex));
103             if (renderIndex)
104             {
105                 sb.Append(link.Index);

```

```

99         sb.Append(':');
100     }
101     if (link.Source == link.Index)
102     {
103         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,
115                 ↪ appendElement, renderIndex);
116         }
117     }
118     sb.Append(' ');
119     if (link.Target == link.Index)
120     {
121         sb.Append(link.Index);
122     }
123     else
124     {
125         var target = new UInt64Link(links.GetLink(link.Target));
126         if (isElement(target))
127         {
128             appendElement(sb, target);
129         }
130         else
131         {
132             links.AppendStructure(sb, visited, target.Index, isElement,
133                 ↪ appendElement, renderIndex);
134         }
135     }
136     sb.Append(')');
137 }
138 else
139 {
140     if (renderDebug)
141     {
142         sb.Append('*');
143     }
144     sb.Append(linkIndex);
145 }
146 }
147 else
148 {
149     if (renderDebug)
150     {
151         sb.Append('~');
152     }
153     sb.Append(linkIndex);
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         /// }

```

```

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 = StructureHelpers.SizeOf<Transition>();
85
86         public readonly ulong TransactionId;
87         public readonly UInt64Link Before;
88         public readonly UInt64Link After;
89         public readonly Timestamp Timestamp;
90
91         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId,
92             ↪ UInt64Link before, UInt64Link after)
93         {
94             TransactionId = transactionId;
95             Before = before;
96             After = after;
97             Timestamp = uniqueTimestampFactory.Create();
98         }
99
100        public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId,
101            ↪ UInt64Link before)
102            : this(uniqueTimestampFactory, transactionId, before, default)
103        {
104        }
105
106        public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
107            : this(uniqueTimestampFactory, transactionId, default, default)
108        {
109        }
110
111        public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
112            ↪ {After}";
113    }

```

```

111 /// <remarks>
112 /// Другие варианты реализации транзакций (атомарности):
113 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker Target))
114 ///     ↳ и индексов.
115 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
116 ///     ↳ потребуется решить вопрос
117 ///     ↳ со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
118 ///     ↳ пересечениями идентификаторов.
119 ///
120 /// Где хранить промежуточный список транзакций?
121 ///
122 /// В оперативной памяти:
123 /// Минусы:
124 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
125 /// так как нужно отдельно выделять память под список трансформаций.
126 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
127 /// если транзакция использует слишком много трансформаций.
128 ///     ↳ -> Можно использовать жёсткий диск для слишком длинных транзакций.
129 ///     ↳ -> Максимальный размер списка трансформаций можно ограничить / задать
130 ///     ↳ константой.
131 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
132 ///     ↳ создавая задержку.
133 ///
134 /// На жёстком диске:
135 /// Минусы:
136 /// 1. Длительный отклик, на запись каждой трансформации.
137 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
138 ///     ↳ -> Это может решаться упаковкой/исключением дублирующих операций.
139 ///     ↳ -> Также это может решаться тем, что короткие транзакции вообще
140 ///     ↳ не будут записываться в случае отката.
141 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
142 ///     ↳ операции (трансформации)
143 ///     ↳ будут записаны в лог.
144 /// </remarks>
145 public class Transaction : DisposableBase
146 {
147     private readonly Queue<Transition> _transitions;
148     private readonly UInt64LinksTransactionsLayer _layer;
149     public bool IsCommitted { get; private set; }
150     public bool IsReverted { get; private set; }
151
152     public Transaction(UInt64LinksTransactionsLayer layer)
153     {
154         _layer = layer;
155         if (_layer._currentTransactionId != 0)
156         {
157             throw new NotSupportedException("Nested transactions not supported.");
158         }
159         IsCommitted = false;
160         IsReverted = false;
161         _transitions = new Queue<Transition>();
162         SetCurrentTransaction(layer, this);
163     }
164
165     public void Commit()
166     {
167         EnsureTransactionAllowsWriteOperations(this);
168         while (_transitions.Count > 0)
169         {
170             var transition = _transitions.Dequeue();
171             _layer._transitions.Enqueue(transition);
172         }
173         _layer._lastCommittedTransactionId = _layer._currentTransactionId;
174         IsCommitted = true;
175     }
176
177     private void Revert()
178     {
179         EnsureTransactionAllowsWriteOperations(this);
180         var transitionsToRevert = new Transition[_transitions.Count];
181         _transitions.CopyTo(transitionsToRevert, 0);
182         for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
183         {
184             _layer.RevertTransition(transitionsToRevert[i]);
185         }
186         IsReverted = true;
187     }
188
189     public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
190     ↳ Transaction transaction)
191     {
192         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
193         layer._currentTransactionTransitions = transaction._transitions;
194         layer._currentTransaction = transaction;
195     }

```

```

191 public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
192 {
193     if (transaction.IsReverted)
194     {
195         throw new InvalidOperationException("Transation is reverted.");
196     }
197     if (transaction.IsCommitted)
198     {
199         throw new InvalidOperationException("Transation is committed.");
200     }
201 }
202
203 protected override void DisposeCore(bool manual, bool wasDisposed)
204 {
205     if (!wasDisposed && _layer != null && !_layer.IsDisposed)
206     {
207         if (!IsCommitted && !IsReverted)
208         {
209             Revert();
210         }
211         _layer.ResetCurrentTransation();
212     }
213 }
214
215 // TODO: THIS IS EXCEPTION WORKAROUND, REMOVE IT THEN
216 ↪ https://github.com/linksplatform/Disposables/issues/13 FIXED
217 protected override bool AllowMultipleDisposeCalls => true;
218
219
220 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
221
222 private readonly string _logAddress;
223 private readonly FileStream _log;
224 private readonly Queue<Transition> _transitions;
225 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
226 private Task _transitionsPusher;
227 private Transition _lastCommittedTransition;
228 private ulong _currentTransactionId;
229 private Queue<Transition> _currentTransactionTransitions;
230 private Transaction _currentTransaction;
231 private ulong _lastCommittedTransactionId;
232
233 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
234 : base(links)
235 {
236     if (string.IsNullOrEmpty(logAddress))
237     {
238         throw new ArgumentNullException(nameof(logAddress));
239     }
240     // В первой строке файла хранится последняя закоммиченную транзакцию.
241     // При запуске это используется для проверки удачного закрытия файла лога.
242     // In the first line of the file the last committed transaction is stored.
243     // On startup, this is used to check that the log file is successfully closed.
244     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
245     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
246     if (!lastCommittedTransition.Equals(lastWrittenTransition))
247     {
248         Dispose();
249         throw new NotSupportedException("Database is damaged, autorecovery is not
250             ↪ supported yet.");
251     }
252     if (lastCommittedTransition.Equals(default(Transition)))
253     {
254         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
255     }
256     _lastCommittedTransition = lastCommittedTransition;
257     // TODO: Think about a better way to calculate or store this value
258     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
259     _lastCommittedTransactionId = allTransitions.Max(x => x.TransactionId);
260     _uniqueTimestampFactory = new UniqueTimestampFactory();
261     _logAddress = logAddress;
262     _log = FileHelpers.Append(logAddress);
263     _transitions = new Queue<Transition>();
264     _transitionsPusher = new Task(TransitionsPusher);
265     _transitionsPusher.Start();
266 }
267
268 public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
269
270 public override ulong Create()
271 {
272     var createdLinkIndex = Links.Create();
273     var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
274     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
275         ↪ default, createdLink));
276     return createdLinkIndex;

```

```

275     }
276
277     public override ulong Update(IList<ulong> parts)
278     {
279         var beforeLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
280         parts[Constants.IndexPart] = Links.Update(parts);
281         var afterLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
282         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
283             ↪ beforeLink, afterLink));
284         return parts[Constants.IndexPart];
285     }
286
287     public override void Delete(ulong link)
288     {
289         var deletedLink = new UInt64Link(Links.GetLink(link));
290         Links.Delete(link);
291         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
292             ↪ deletedLink, default));
293     }
294
295     [MethodImpl(MethodImplOptions.AggressiveInlining)]
296     private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
297         ↪ _transitions;
298
299     private void CommitTransition(Transition transition)
300     {
301         if (_currentTransaction != null)
302         {
303             Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
304         }
305         var transitions = GetCurrentTransitions();
306         transitions.Enqueue(transition);
307     }
308
309     private void RevertTransition(Transition transition)
310     {
311         if (transition.After.IsNull()) // Revert Deletion with Creation
312         {
313             Links.Create();
314         }
315         else if (transition.Before.IsNull()) // Revert Creation with Deletion
316         {
317             Links.Delete(transition.After.Index);
318         }
319         else // Revert Update
320         {
321             Links.Update(new[] { transition.After.Index, transition.Before.Source,
322                 ↪ transition.Before.Target });
323         }
324     }
325
326     private void ResetCurrentTransation()
327     {
328         _currentTransactionId = 0;
329         _currentTransactionTransitions = null;
330         _currentTransaction = null;
331     }
332
333     private void PushTransitions()
334     {
335         if (_log == null || _transitions == null)
336         {
337             return;
338         }
339         for (var i = 0; i < _transitions.Count; i++)
340         {
341             var transition = _transitions.Dequeue();
342             _log.Write(transition);
343             _lastCommittedTransition = transition;
344         }
345     }
346
347     private void TransitionsPusher()
348     {
349         while (!IsDisposed && _transitionsPusher != null)
350         {
351             Thread.Sleep(DefaultPushDelay);
352             PushTransitions();
353         }
354     }
355
356     public Transaction BeginTransaction() => new Transaction(this);
357
358     private void DisposeTransitions()
359     {
360         try

```

```

358     {
359         var pusher = _transitionsPusher;
360         if (pusher != null)
361         {
362             _transitionsPusher = null;
363             pusher.Wait();
364         }
365         if (_transitions != null)
366         {
367             PushTransitions();
368         }
369         Disposable.TryDispose(_log);
370         FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
371     }
372     catch
373     {
374     }
375 }
376
377 #region DisposalBase
378
379 protected override void DisposeCore(bool manual, bool wasDisposed)
380 {
381     if (!wasDisposed)
382     {
383         DisposeTransitions();
384     }
385     base.DisposeCore(manual, wasDisposed);
386 }
387
388 #endregion
389 }
390

```

Index

- ./Converters/AddressToUnaryNumberConverter.cs, 1
- ./Converters/LinkToltsFrequencyNumberConveter.cs, 1
- ./Converters/PowerOf2ToUnaryNumberConverter.cs, 2
- ./Converters/UnaryNumberToAddressAddOperationConverter.cs, 2
- ./Converters/UnaryNumberToAddressOrOperationConverter.cs, 3
- ./Decorators/LinksCascadeDependenciesResolver.cs, 3
- ./Decorators/LinksCascadeUniquenessAndDependenciesResolver.cs, 4
- ./Decorators/LinksDecoratorBase.cs, 4
- ./Decorators/LinksDependenciesValidator.cs, 5
- ./Decorators/LinksDisposableDecoratorBase.cs, 5
- ./Decorators/LinksInnerReferenceValidator.cs, 5
- ./Decorators/LinksNonExistentReferencesCreator.cs, 6
- ./Decorators/LinksNullToSelfReferenceResolver.cs, 6
- ./Decorators/LinksSelfReferenceResolver.cs, 7
- ./Decorators/LinksUniquenessResolver.cs, 7
- ./Decorators/LinksUniquenessValidator.cs, 7
- ./Decorators/NonNullContentsLinkDeletionResolver.cs, 8
- ./Decorators/UInt64Links.cs, 8
- ./Decorators/UniLinks.cs, 9
- ./Doublet.cs, 14
- ./DoubletComparer.cs, 13
- ./Hybrid.cs, 14
- ./ILinks.cs, 15
- ./ILinksExtensions.cs, 15
- ./ISynchronizedLinks.cs, 25
- ./Incrementers/FrequencyIncrementer.cs, 23
- ./Incrementers/LinkFrequencyIncrementer.cs, 24
- ./Incrementers/UnaryNumberIncrementer.cs, 24
- ./Link.cs, 25
- ./LinkExtensions.cs, 27
- ./LinksOperatorBase.cs, 27
- ./PropertyOperators/DefaultLinkPropertyOperator.cs, 27
- ./PropertyOperators/FrequencyPropertyOperator.cs, 28
- ./ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs, 36
- ./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs, 37
- ./ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 29
- ./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs, 49
- ./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs, 49
- ./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 43
- ./Sequences/Converters/BalancedVariantConverter.cs, 55
- ./Sequences/Converters/CompressingConverter.cs, 56
- ./Sequences/Converters/LinksListToSequenceConverterBase.cs, 58
- ./Sequences/Converters/OptimalVariantConverter.cs, 59
- ./Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 60
- ./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs, 60
- ./Sequences/CreteriaMatchers/MarkedSequenceCreteriaMatcher.cs, 60
- ./Sequences/DefaultSequenceAppender.cs, 61
- ./Sequences/DuplicateSegmentsCounter.cs, 61
- ./Sequences/DuplicateSegmentsProvider.cs, 62
- ./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs, 64
- ./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToltsFrequencyNumberConverter.cs, 64
- ./Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 64
- ./Sequences/Frequencies/Cache/LinkFrequency.cs, 66
- ./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 66
- ./Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 66
- ./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 67
- ./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 67
- ./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 67
- ./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 67
- ./Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 68
- ./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 69
- ./Sequences/HeightProviders/ISequenceHeightProvider.cs, 69
- ./Sequences/Sequences.Experiments.ReadSequence.cs, 101
- ./Sequences/Sequences.Experiments.cs, 78
- ./Sequences/Sequences.cs, 69
- ./Sequences/SequencesExtensions.cs, 102
- ./Sequences/SequencesIndexer.cs, 103
- ./Sequences/SequencesOptions.cs, 104
- ./Sequences/UnicodeMap.cs, 105
- ./Sequences/Walkers/LeftSequenceWalker.cs, 107
- ./Sequences/Walkers/RightSequenceWalker.cs, 108
- ./Sequences/Walkers/SequenceWalkerBase.cs, 108

./Stacks/Stack.cs, 109
./Stacks/StackExtensions.cs, 109
./SynchronizedLinks.cs, 109
./UInt64Link.cs, 110
./UInt64LinkExtensions.cs, 112
./UInt64LinksExtensions.cs, 112
./UInt64LinksTransactionsLayer.cs, 114
./obj/Debug/netstandard2.0/Platform.Data.Doublets.AssemblyInfo.cs, 27