

## 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>,
9         ↳ IConverter<TLink>
10    {
11        private static readonly EqualityComparer<TLink> _equalityComparer =
12            ↳ EqualityComparer<TLink>.Default;
13
14        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
15
16        public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int,
17            ↳ TLink> powerOf2ToUnaryNumberConverter) : base(links) =>
18            ↳ _powerOf2ToUnaryNumberConverter = powerOf2ToUnaryNumberConverter;
19
20        public TLink Convert(TLink sourceAddress)
21        {
22            var number = sourceAddress;
23            var target = Links.Constants.Null;
24            for (int i = 0; i < CachedTypeInfo<TLink>.BitsLength; i++)
25            {
26                if (_equalityComparer.Equals(ArithmeticHelpers.And(number,
27                    ↳ Integer<TLink>.One), Integer<TLink>.One))
28                {
29                    target = _equalityComparer.Equals(target, Links.Constants.Null)
30                        ? _powerOf2ToUnaryNumberConverter.Convert(i)
31                        : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i),
32                            ↳ target);
33                }
34                number = (Integer<TLink>)((ulong)(Integer<TLink>)number >> 1); //
35                ↳ Should be BitwiseHelpers.ShiftRight(number, 1);
36                if (_equalityComparer.Equals(number, default))
37                {
38                    break;
39                }
40            }
41            return target;
42        }
43    }
44 }
```

### ./Converters/LinkToItsFrequencyNumberConveter.cs

```
1 using System;
2 using System.Collections.Generic;
3 using Platform.Interfaces;
4
5 namespace Platform.Data.Doublets.Converters
6 {
7     public class LinkToItsFrequencyNumberConveter<TLink> :
8         ↳ LinksOperatorBase<TLink>, IConverter<Doublet<TLink>, TLink>
9    {
10        private static readonly EqualityComparer<TLink> _equalityComparer =
11            ↳ EqualityComparer<TLink>.Default;
12
13        private readonly ISpecificPropertyOperator<TLink, TLink>
14            ↳ _frequencyPropertyOperator;
```

```
12 private readonly IConverter<TLink> _unaryNumberToAddressConverter;
13
14 public LinkToItsFrequencyNumberConveter(
15     ILinks<TLink> links,
16     ISpecificPropertyOperator<TLink, TLink> frequencyPropertyOperator,
17     IConverter<TLink> unaryNumberToAddressConverter)
18     : base(links)
19 {
20     _frequencyPropertyOperator = frequencyPropertyOperator;
21     _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
22 }
23
24 public TLink Convert(Doublet<TLink> doublet)
25 {
26     var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
27     if (_equalityComparer.Equals(link, Links.Constants.Null))
28     {
29         throw new ArgumentException($"Link with {doublet.Source} source and
30             ↳ {doublet.Target} target not found.", nameof(doublet));
31     }
32     var frequency = _frequencyPropertyOperator.Get(link);
33     if (_equalityComparer.Equals(frequency, default))
34     {
35         return default;
36     }
37     var frequencyNumber = Links.GetSource(frequency);
38     var number = _unaryNumberToAddressConverter.Convert(frequencyNumber);
39     return number;
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>,
8         ↳ IConverter<int, TLink>
9    {
10        private static readonly EqualityComparer<TLink> _equalityComparer =
11            ↳ EqualityComparer<TLink>.Default;
12
13        private readonly TLink[] _unaryNumberPowersOf2;
14
15        public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) :
16            ↳ base(links)
17        {
18            _unaryNumberPowersOf2 = new TLink[64];
19            _unaryNumberPowersOf2[0] = one;
20        }
21
22        public TLink Convert(int power)
23        {
24            if (power < 0 || power >= _unaryNumberPowersOf2.Length)
25            {
26                throw new ArgumentOutOfRangeException(nameof(power));
27            }
28            if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
```

```

26     {
27         return _unaryNumberPowersOf2[power];
28     }
29     var previousPowerOf2 = Convert(power - 1);
30     var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
31     _unaryNumberPowersOf2[power] = powerOf2;
32     return powerOf2;
33 }
34 }
35 }

```

## ./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> :
8          ↳ LinksOperatorBase<TLink>, 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
17             ↳ _unaryOne)
18             : base(links)
19         {
20             _unaryOne = unaryOne;
21             InitUnaryToUInt64();
22         }
23
24         private void InitUnaryToUInt64()
25         {
26             _unaryToUInt64 = new Dictionary<TLink, TLink>
27             {
28                 { _unaryOne, Integer<TLink>.One }
29             };
30             var unary = _unaryOne;
31             var number = Integer<TLink>.One;
32             for (var i = 1; i < 64; i++)
33             {
34                 _unaryToUInt64.Add(unary = Links.GetOrCreate(unary, unary), number =
35                     ↳ (Integer<TLink>)((Integer<TLink>)number * 2UL));
36             }
37         }
38
39         public TLink Convert(TLink unaryNumber)
40         {
41             if (_equalityComparer.Equals(unaryNumber, default))
42             {
43                 return default;
44             }
45             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
46             {
47                 return Integer<TLink>.One;
48             }
49             var source = Links.GetSource(unaryNumber);
50             var target = Links.GetTarget(unaryNumber);
51             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> :
9          ↳ LinksOperatorBase<TLink>, 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,
17             ↳ IConverter<int, 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.Con
24                     ↳ vert(i),
25                     ↳ i);
26             }
27         }
28
29         public TLink Convert(TLink sourceNumber)
30         {
31             var source = sourceNumber;
32             var target = Links.Constants.Null;
33             while (!_equalityComparer.Equals(source, Links.Constants.Null))
34             {
35                 if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int
36                     ↳ powerOf2Index))
37                 {
38                     source = Links.Constants.Null;
39                 }
40                 else

```

```

35         {
36             powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
37             source = Links.GetTarget(source);
38         }
39         target = (Integer<TLink>)((Integer<TLink>)target | 1UL << powerOf2Index);
40         ↪ // MathHelpers.Or(target, MathHelpers.ShiftLeft(One, powerOf2Index))
41     }
42     return target;
43 }
44 }

```

#### ./Decorators/LinksCascadeDependenciesResolver.cs

```

1  using System.Collections.Generic;
2  using Platform.Collections.Arrays;
3  using Platform.Numbers;
4
5  namespace Platform.Data.Doublets.Decorators
6  {
7      public class LinksCascadeDependenciesResolver<TLink> : LinksDecoratorBase<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↪ EqualityComparer<TLink>.Default;
11
12         public LinksCascadeDependenciesResolver(ILinks<TLink> links) : base(links) { }
13
14         public override void Delete(TLink link)
15         {
16             EnsureNoDependenciesOnDelete(link);
17             base.Delete(link);
18         }
19
20         public void EnsureNoDependenciesOnDelete(TLink link)
21         {
22             ulong referencesCount = (Integer<TLink>)Links.Count(Constants.Any, link);
23             var references = ArrayPool.Allocate<TLink>((long)referencesCount);
24             var referencesFiller = new ArrayFiller<TLink, TLink>(references,
25                 ↪ Constants.Continue);
26             Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, link);
27             //references.Sort() // TODO: Решить необходимо ли для корректного порядка
28             ↪ отмены операций в транзакциях
29             for (var i = (long)referencesCount - 1; i >= 0; i--)
30             {
31                 if (_equalityComparer.Equals(references[i], link))
32                 {
33                     continue;
34                 }
35                 Links.Delete(references[i]);
36             }
37             ArrayPool.Free(references);
38         }
39     }
40 }

```

#### ./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>

```

```

8  {
9      private static readonly EqualityComparer<TLink> _equalityComparer =
10         ↪ EqualityComparer<TLink>.Default;
11
12     public LinksCascadeUniquenessAndDependenciesResolver(ILinks<TLink> links) :
13         ↪ base(links) { }
14
15     protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress,
16         ↪ TLink 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,
26             ↪ Constants.Continue);
27         Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any,
28             ↪ oldLinkAddress, Constants.Any);
29         Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any,
30             ↪ Constants.Any, oldLinkAddress);
31         for (ulong i = 0; i < referencesAsSourceCount; i++)
32         {
33             var reference = references[i];
34             if (!_equalityComparer.Equals(reference, oldLinkAddress))
35             {
36                 Links.Update(reference, newLinkAddress, Links.GetTarget(reference));
37             }
38         }
39         for (var i = (long)referencesAsSourceCount; i < references.Length; i++)
40         {
41             var reference = references[i];
42             if (!_equalityComparer.Equals(reference, oldLinkAddress))
43             {
44                 Links.Update(reference, Links.GetSource(reference), newLinkAddress);
45             }
46         }
47         ArrayPool.Free(references);
48         return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
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);

```

```

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
30     protected override bool AllowMultipleDisposeCalls => true;
31
32     protected override void DisposeCore(bool manual, bool wasDisposed) =>
33         ↳ Disposable.TryDispose(Links);
34 }

```

#### ./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
7     ↳ or to be external (hybrid link's raw number).
8     public class LinksInnerReferenceValidator<T> : LinksDecoratorBase<T>
9     {
10         public LinksInnerReferenceValidator(ILinks<T> links) : base(links) { }
11
12         public override T Each(Func<IList<T>, T> handler, IList<T> restrictions)
13         {
14             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
15             return base.Each(handler, restrictions);
16         }
17
18         public override T Count(IList<T> restriction)
19         {
20             Links.EnsureInnerReferenceExists(restriction, nameof(restriction));
21             return base.Count(restriction);
22         }
23
24         public override T Update(IList<T> restrictions)
25         {
26             // TODO: Possible values: null, ExistentLink or
27             ↳ NonExistentHybrid(ExternalReference)
28             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
29             return base.Update(restrictions);
30         }
31
32         public override void Delete(T link)
33         {
34             // TODO: Решить считать ли такое исключением, или лишь более конкретным
35             ↳ требованием?
36             Links.EnsureLinkExists(link, nameof(link));
37             base.Delete(link);
38         }
39     }
40 }

```

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

```

#### ./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 =
    ↪ EqualityComparer<TLink>.Default;
8
9           public LinksNullToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
10
11           public override TLink Create()
12           {
13               var link = base.Create();
14               return Links.Update(link, link, link);
15           }
16
17           public override TLink Update(IList<TLink> restrictions)
18           {
19               restrictions[Constants.SourcePart] =
    ↪ _equalityComparer.Equals(restrictions[Constants.SourcePart],
    ↪ Constants.Null) ? restrictions[Constants.IndexPart] :
    ↪ restrictions[Constants.SourcePart];
20               restrictions[Constants.TargetPart] =
    ↪ _equalityComparer.Equals(restrictions[Constants.TargetPart],
    ↪ Constants.Null) ? restrictions[Constants.IndexPart] :
    ↪ restrictions[Constants.TargetPart];
21               return base.Update(restrictions);
22           }
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 =
    ↪ EqualityComparer<TLink>.Default;

```

```

9       public LinksSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
10
11       public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink>
    ↪ restrictions)
12       {
13           if (!_equalityComparer.Equals(Constants.Any, Constants.Itself)
14               && (((restrictions.Count > Constants.IndexPart) &&
    ↪ _equalityComparer.Equals(restrictions[Constants.IndexPart],
    ↪ Constants.Itself))
15                   || ((restrictions.Count > Constants.SourcePart) &&
    ↪ _equalityComparer.Equals(restrictions[Constants.SourcePart],
    ↪ Constants.Itself))
16                   || ((restrictions.Count > Constants.TargetPart) &&
    ↪ _equalityComparer.Equals(restrictions[Constants.TargetPart],
    ↪ Constants.Itself))))
17           {
18               return Constants.Continue;
19           }
20           return base.Each(handler, restrictions);
21       }
22
23       public override TLink Update(IList<TLink> restrictions)
24       {
25           restrictions[Constants.SourcePart] =
    ↪ _equalityComparer.Equals(restrictions[Constants.SourcePart],
    ↪ Constants.Itself) ? restrictions[Constants.IndexPart] :
    ↪ restrictions[Constants.SourcePart];
26           restrictions[Constants.TargetPart] =
    ↪ _equalityComparer.Equals(restrictions[Constants.TargetPart],
    ↪ Constants.Itself) ? restrictions[Constants.IndexPart] :
    ↪ restrictions[Constants.TargetPart];
27           return base.Update(restrictions);
28       }
29   }
30 }
31 }

```

#### ./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 =
    ↪ EqualityComparer<TLink>.Default;
8
9           public LinksUniquenessResolver(ILinks<TLink> links) : base(links) { }
10
11           public override TLink Update(IList<TLink> restrictions)
12           {
13               var newLinkAddress = Links.SearchOrDefault(restrictions[Constants.SourcePart],
    ↪ restrictions[Constants.TargetPart]);
14               if (_equalityComparer.Equals(newLinkAddress, default))
15               {
16                   return base.Update(restrictions);
17               }
18               return ResolveAddressChangeConflict(restrictions[Constants.IndexPart],
    ↪ newLinkAddress);
19           }

```

```

20
21     protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
    ↪     newLinkAddress)
22     {
23         if (Links.Exists(oldLinkAddress))
24         {
25             Delete(oldLinkAddress);
26         }
27         return newLinkAddress;
28     }
29 }
30 }

```

#### ./Decorators/LinksUniquenessValidator.cs

```

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

```

#### ./Decorators/NonNullContentsLinkDeletionResolver.cs

```

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

```

#### ./Decorators/UInt64Links.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Collections;
4  using Platform.Collections.Arrays;
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      /// <summary>
9      /// Представляет объект для работы с базой данных (файлом) в формате Links
    ↪     (массива взаимосвязей).
10     /// </summary>
11     /// <remarks>
12     /// Возможные оптимизации:
13     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
14     /// + меньше объём БД

```

```

15     /// - меньше производительность
16     /// - больше ограничение на количество связей в БД)
17     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования
    ↪     БД)
18     /// + меньше объём БД
19     /// - больше сложность
20
21     /// AVL - высота дерева может позволить точно рассчитать размер дерева, нет
    ↪     необходимости в SBT.
22     /// AVL дерево можно прошить.
23
24     /// Текущее теоретическое ограничение на размер связей - long.MaxValue
25     /// Желательно реализовать поддержку переключения между деревьями и битовыми
    ↪     индексами (битовыми строками) - вариант матрицы (выстраиваемой лениво).
26     ///
27     /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения
    ↪     будут выбрасываться только при #if DEBUG
28     </remarks>
29     public class UInt64Links : LinksDisposableDecoratorBase<ulong>
30     {
31         public UInt64Links(ILinks<ulong> links) : base(links) { }
32
33         public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong>
    ↪         restrictions)
34         {
35             this.EnsureLinkIsAnyOrExists(restrictions);
36             return Links.Each(handler, restrictions);
37         }
38
39         public override ulong Create() => Links.CreatePoint();
40
41         public override ulong Update(IList<ulong> restrictions)
42         {
43             if (restrictions.IsNullOrEmpty())
44             {
45                 return Constants.Null;
46             }
47             // TODO: Remove usages of these hacks (these should not be backwards compatible)
48             if (restrictions.Count == 2)
49             {
50                 return this.Merge(restrictions[0], restrictions[1]);
51             }
52             if (restrictions.Count == 4)
53             {
54                 return this.UpdateOrCreateOrGet(restrictions[0], restrictions[1], restrictions[2],
    ↪                 restrictions[3]);
55             }
56             // TODO: Looks like this is a common type of exceptions linked with restrictions
    ↪             support
57             if (restrictions.Count != 3)
58             {
59                 throw new NotSupportedException();
60             }
61             var updatedLink = restrictions[Constants.IndexPart];
62             this.EnsureLinkExists(updatedLink, nameof(Constants.IndexPart));
63             var newSource = restrictions[Constants.SourcePart];
64             this.EnsureLinkIsItselfOrExists(newSource, nameof(Constants.SourcePart));
65             var newTarget = restrictions[Constants.TargetPart];
66             this.EnsureLinkIsItselfOrExists(newTarget, nameof(Constants.TargetPart));
67             var existedLink = Constants.Null;
68             if (newSource != Constants.Itself && newTarget != Constants.Itself)

```

```

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

```

## ./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
            ↪ Default UniLinksBase, 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
59  ↳ (they are also Null by definition)
60  ///     var matchDecision = matchedHandler(, NullLink);
61  ///     if (Equals(matchDecision, Constants.Break))
62  ///         return false;
63  ///     if (!Equals(matchDecision, Constants.Skip))
64  ///         transitions.Add(new Transition(matchedLink, newValue));
65  ///     }
66  ///     else
67  ///     {
68  ///         Func<T, bool> handler;
69  ///         handler = link =>
70  ///         {
71  ///             var matchedLink = Memory.GetLinkValue(link);
72  ///             var newValue = Memory.GetLinkValue(link);
73  ///             newValue[Constants.IndexPart] = Constants.Itself;
74  ///             newValue[Constants.SourcePart] =
75  /// ↳ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
76  /// ↳ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
77  ///             newValue[Constants.TargetPart] =
78  /// ↳ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
79  /// ↳ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
80  ///             var matchDecision = matchedHandler(matchedLink, newValue);
81  ///             if (Equals(matchDecision, Constants.Break))
82  ///                 return false;
83  ///             if (!Equals(matchDecision, Constants.Skip))
84  ///                 transitions.Add(new Transition(matchedLink, newValue));
85  ///             return true;
86  ///         };
87  ///         if (!Memory.Each(handler, restriction))
88  ///             return Constants.Break;
89  ///     }
90  /// }
91  /// else
92  /// {
93  ///     Func<T, bool> handler = link =>
94  ///     {
95  ///         var matchedLink = Memory.GetLinkValue(link);
96  ///         var matchDecision = matchedHandler(matchedLink, matchedLink);
97  ///         return !Equals(matchDecision, Constants.Break);
98  ///     };
99  ///     if (!Memory.Each(handler, restriction))
100  ///         return Constants.Break;
101  /// }
102  /// }
103  /// else
104  /// {
105  ///     if (substitution != null)
106  ///     {
107  ///         transitions = new List<ILink<T>>();
108  ///         Func<T, bool> handler = link =>
109  ///         {
110  ///             var matchedLink = Memory.GetLinkValue(link);
111  ///             transitions.Add(matchedLink);
112  ///             return true;
113  ///         };
114  ///         if (!Memory.Each(handler, restriction))
115  ///             return Constants.Break;
116  ///     }
117  ///     else
118  ///     {

```

```

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 =>
145  /// ↳ !Equals(matchedHandler(Memory.GetLinkValue(link)), Constants.Break),
146  /// ↳ restriction))
147  ///         return Constants.Break;
148  }
149  else // Update
150  {
151  ///     //List<ILink<T>> matchedLinks = null;
152  ///     if (matchedHandler != null)
153  ///     {
154  ///         matchedLinks = new List<ILink<T>>();
155  ///         Func<T, bool> handler = link =>
156  ///         {
157  ///             var matchedLink = Memory.GetLinkValue(link);
158  ///             var matchDecision = matchedHandler(matchedLink);
159  ///             if (Equals(matchDecision, Constants.Break))
160  ///                 return false;
161  ///             if (!Equals(matchDecision, Constants.Skip))
162  ///                 matchedLinks.Add(matchedLink);
163  ///             return true;
164  ///         };
165  ///         if (!Memory.Each(handler, restriction))
166  ///             return Constants.Break;
167  ///     }
168  if (!matchedLinks.IsNullOrEmpty())
169  {
170  var totalMatchedLinks = matchedLinks.Count;
171  for (var i = 0; i < totalMatchedLinks; i++)
172  {

```



```

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

```

```

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

```

```

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

```

## ./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 =
            ↪ EqualityComparer<T>.Default;
13

```

```

14     public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     public bool Equals(Doublet<T> x, Doublet<T> y) =>
            ↪ _equalityComparer.Equals(x.Source, y.Source) &&
            ↪ _equalityComparer.Equals(x.Target, y.Target);
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     public int GetHashCode(Doublet<T> obj) => unchecked(obj.Source.GetHashCode()
            ↪ << 15 ^ obj.Target.GetHashCode());
21 }
22 }

```

## ./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 =
            ↪ EqualityComparer<T>.Default;
9
10         public T Source { get; set; }
11         public T Target { get; set; }
12
13         public Doublet(T source, T target)
14         {
15             Source = source;
16             Target = target;
17         }
18
19         public override string ToString() => $"{Source}->{Target}";
20
21         public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source,
            ↪ other.Source) && _equalityComparer.Equals(Target, other.Target);
22     }
23 }

```

## ./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,
    ↳ CachedTypeInfo<T>.SignedVersion));
27
28 public Hybrid(object value, bool isExternal)
29 {
30     var signedType = CachedTypeInfo<T>.SignedVersion;
31     var signedValue = Convert.ChangeType(value, signedType);
32     var abs = typeof(MathHelpers).GetTypeInfo().GetMethod("Abs").MakeGenericM
    ↳ ethod(signedType);
33     var negate = typeof(MathHelpers).GetTypeInfo().GetMethod("Negate").MakeGen
    ↳ ericMethod(signedType);
34     var absoluteValue = abs.Invoke(null, new[] { signedValue });
35     var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
    ↳ absoluteValue;
36     Value = To.UnsignedAs<T>(resultValue);
37 }
38
39 public static implicit operator Hybrid<T>(T integer) => new Hybrid<T>(integer);
40
41 public static explicit operator Hybrid<T>(ulong integer) => new Hybrid<T>(integer);
42
43 public static explicit operator Hybrid<T>(long integer) => new Hybrid<T>(integer);
44
45 public static explicit operator Hybrid<T>(uint integer) => new Hybrid<T>(integer);
46
47 public static explicit operator Hybrid<T>(int integer) => new Hybrid<T>(integer);
48
49 public static explicit operator Hybrid<T>(ushort integer) => new
    ↳ Hybrid<T>(integer);
50
51 public static explicit operator Hybrid<T>(short integer) => new Hybrid<T>(integer);
52
53 public static explicit operator Hybrid<T>(byte integer) => new Hybrid<T>(integer);
54
55 public static explicit operator Hybrid<T>(sbyte integer) => new Hybrid<T>(integer);
56
57 public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;
58
59 public static explicit operator ulong(Hybrid<T> hybrid) =>
    ↳ Convert.ToUInt64(hybrid.Value);
60
61 public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;
62
63 public static explicit operator uint(Hybrid<T> hybrid) =>
    ↳ Convert.ToUInt32(hybrid.Value);
64
65 public static explicit operator int(Hybrid<T> hybrid) =>
    ↳ Convert.ToInt32(hybrid.AbsoluteValue);
66
67 public static explicit operator ushort(Hybrid<T> hybrid) =>
    ↳ Convert.ToUInt16(hybrid.Value);
68
69 public static explicit operator short(Hybrid<T> hybrid) =>
    ↳ Convert.ToInt16(hybrid.AbsoluteValue);
70
71 public static explicit operator byte(Hybrid<T> hybrid) =>
    ↳ Convert.ToByte(hybrid.Value);
72

```

```

73     public static explicit operator sbyte(Hybrid<T> hybrid) =>
    ↳ Convert.ToSByte(hybrid.AbsoluteValue);
74
75     public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
    ↳ default(T).ToString() : IsExternal ? §"<{AbsoluteValue}>" : Value.ToString();
76 }
77 }
78
79 ./ILinks.cs
80
81 using Platform.Data.Constants;
82
83 namespace Platform.Data.Doublets
84 {
85     public interface ILinks<TLink> : ILinks<TLink, LinksCombinedConstants<TLink,
    ↳ TLink, int>>
86     {
87     }
88 }
89
90 ./ILinksExtensions.cs
91
92 using System;
93 using System.Collections;
94 using System.Collections.Generic;
95 using System.Linq;
96 using System.Runtime.CompilerServices;
97 using Platform.Ranges;
98 using Platform.Collections.Arrays;
99 using Platform.Numbers;
100 using Platform.Random;
101 using Platform.Helpers.Setters;
102 using Platform.Data.Exceptions;
103
104 namespace Platform.Data.Doublets
105 {
106     public static class ILinksExtensions
107     {
108         public static void RunRandomCreations<TLink>(this ILinks<TLink> links, long
    ↳ amountOfCreations)
109         {
110             for (long i = 0; i < amountOfCreations; i++)
111             {
112                 var linksAddressRange = new Range<ulong>(0, (Integer<TLink>)links.Count());
113                 Integer<TLink> source =
    ↳ RandomHelpers.Default.NextUInt64(linksAddressRange);
114                 Integer<TLink> target =
    ↳ RandomHelpers.Default.NextUInt64(linksAddressRange);
115                 links.CreateAndUpdate(source, target);
116             }
117         }
118
119         public static void RunRandomSearches<TLink>(this ILinks<TLink> links, long
    ↳ amountOfSearches)
120         {
121             for (long i = 0; i < amountOfSearches; i++)
122             {
123                 var linkAddressRange = new Range<ulong>(1, (Integer<TLink>)links.Count());
124                 Integer<TLink> source =
    ↳ RandomHelpers.Default.NextUInt64(linkAddressRange);
125                 Integer<TLink> target =
    ↳ RandomHelpers.Default.NextUInt64(linkAddressRange);
126                 links.SearchOrDefault(source, target);
127             }
128         }
129     }
130 }

```

```

37     }
38
39     public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, long
↳ amountOfDeletions)
40     {
41         var min = (ulong)amountOfDeletions > (Integer<TLink>)links.Count() ? 1 :
↳ (Integer<TLink>)links.Count() - (ulong)amountOfDeletions;
42         for (long i = 0; i < amountOfDeletions; i++)
43         {
44             var linksAddressRange = new Range<ulong>(min,
↳ (Integer<TLink>)links.Count());
45             Integer<TLink> link = RandomHelpers.Default.NextUInt64(linksAddressRange);
46             links.Delete(link);
47             if ((Integer<TLink>)links.Count() < min)
48             {
49                 break;
50             }
51         }
52     }
53
54     /// <remarks>
55     /// TODO: Возможно есть очень простой способ это сделать.
56     /// (Например просто удалить файл, или изменить его размер таким образом,
57     /// чтобы удалился весь контент)
58     /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
59     /// </remarks>
60     public static void DeleteAll<TLink>(this ILinks<TLink> links)
61     {
62         var equalityComparer = EqualityComparer<TLink>.Default;
63         var comparer = Comparer<TLink>.Default;
64         for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
↳ ArithmeticHelpers.Decrement(i))
65         {
66             links.Delete(i);
67             if (!equalityComparer.Equals(links.Count(), ArithmeticHelpers.Decrement(i)))
68             {
69                 i = links.Count();
70             }
71         }
72     }
73
74     public static TLink First<TLink>(this ILinks<TLink> links)
75     {
76         TLink firstLink = default;
77         var equalityComparer = EqualityComparer<TLink>.Default;
78         if (equalityComparer.Equals(links.Count(), default))
79         {
80             throw new Exception("В хранилище нет связей.");
81         }
82         links.Each(links.Constants.Any, links.Constants.Any, link =>
83         {
84             firstLink = link|links.Constants.IndexPart;
85             return links.Constants.Break;
86         });
87         if (equalityComparer.Equals(firstLink, default))
88         {
89             throw new Exception("В процессе поиска по хранилищу не было найдено
↳ связей.");
90         }
91         return firstLink;
92     }
93

```

```

94     public static bool IsInnerReference<TLink>(this ILinks<TLink> links, TLink
↳ reference)
95     {
96         var constants = links.Constants;
97         var comparer = Comparer<TLink>.Default;
98         return comparer.Compare(constants.MinPossibleIndex, reference) >= 0 &&
↳ comparer.Compare(reference, constants.MaxPossibleIndex) <= 0;
99     }
100
101     #region Paths
102
103     /// <remarks>
104     /// TODO: Как так? Как то что ниже может быть корректно?
105     /// Скорее всего практически не применимо
106     /// Предполагалось, что можно было конвертировать формируемый в проходе
↳ через SequenceWalker
107     /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
108     /// TODO: Возможно нужен метод, который именно выбрасывает исключения
↳ (EnsurePathExists)
109     /// </remarks>
110     public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params
↳ TLink[] path)
111     {
112         var current = path[0];
113         //EnsureLinkExists(current, "path");
114         if (!links.Exists(current))
115         {
116             return false;
117         }
118         var equalityComparer = EqualityComparer<TLink>.Default;
119         var constants = links.Constants;
120         for (var i = 1; i < path.Length; i++)
121         {
122             var next = path[i];
123             var values = links.GetLink(current);
124             var source = values|constants.SourcePart;
125             var target = values|constants.TargetPart;
126             if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
↳ next))
127             {
128                 //throw new Exception(string.Format("Невозможно выбрать путь, так как
↳ и Source и Target совпадают с элементом пути {0}.", next));
129                 return false;
130             }
131             if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
↳ target))
132             {
133                 //throw new Exception(string.Format("Невозможно продолжить путь через
↳ элемент пути {0}", next));
134                 return false;
135             }
136             current = next;
137         }
138         return true;
139     }
140
141     /// <remarks>
142     /// Может потребовать дополнительного стека для PathElement's при
↳ использовании SequenceWalker.
143     /// </remarks>

```

```

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

```

```

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

```

```

241 [MethodImpl(MethodImplOptions.AggressiveInlining)]
242 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target, Func<TLink, bool> handler)
243 {
244     var constants = links.Constants;
245     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
    ↳ constants.Break, constants.Any, source, target);
246 }
247
248 /// <summary>
249 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая
    ↳ обработчик (handler) для каждой подходящей связи.
250 /// </summary>
251 /// <param name="links">Хранилище связей.</param>
252 /// <param name="source">Значение, определяющее соответствующие шаблону
    ↳ связи. (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве
    ↳ начала, Constants.Any - любое начало, 1..∞ конкретное начало)</param>
253 /// <param name="target">Значение, определяющее соответствующие шаблону
    ↳ связи. (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве
    ↳ конца, Constants.Any - любой конец, 1..∞ конкретный конец)</param>
254 /// <param name="handler">Обработчик каждой подходящей связи.</param>
255 /// <returns>True, в случае если проход по связям не был прерван и False в
    ↳ обратном случае.</returns>
256 [MethodImpl(MethodImplOptions.AggressiveInlining)]
257 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target, Func<IList<TLink>, TLink> handler)
258 {
259     var constants = links.Constants;
260     return links.Each(handler, constants.Any, source, target);
261 }
262
263 [MethodImpl(MethodImplOptions.AggressiveInlining)]
264 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params
    ↳ TLink[] restrictions)
265 {
266     var constants = links.Constants;
267     int listSize = (Integer<TLink>)links.Count(restrictions);
268     var list = new IList<TLink>[listSize];
269     if (listSize > 0)
270     {
271         var filler = new ArrayFiller<IList<TLink>, TLink>(list,
            ↳ links.Constants.Continue);
272         links.Each(filler.AddAndReturnConstant, restrictions);
273     }
274     return list;
275 }
276
277 /// <summary>
278 /// Возвращает значение, определяющее существует ли связь с указанными
    ↳ началом и концом в хранилище связей.
279 /// </summary>
280 /// <param name="links">Хранилище связей.</param>
281 /// <param name="source">Начало связи.</param>
282 /// <param name="target">Конец связи.</param>
283 /// <returns>Значение, определяющее существует ли связь.</returns>
284 [MethodImpl(MethodImplOptions.AggressiveInlining)]
285 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target) =>
    ↳ Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source,
    ↳ target), default) > 0;

```

```

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, link);
403     if (equalityComparer.Equals(values[constants.TargetPart], link))
404     {
405         referencesAsTarget--;
406     }
407     return referencesAsSource + referencesAsTarget;
408 }
409
410 /// <param name="links">Хранилище связей.</param>
411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
412 public static bool DependenciesExist<TLink>(this ILinks<TLink> links, TLink link)
413     ↳ => links.DependenciesCount(link) > 0;
414
415 /// <param name="links">Хранилище связей.</param>
416 [MethodImpl(MethodImplOptions.AggressiveInlining)]
417 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink
    ↳ source, TLink target)
418 {
419     var constants = links.Constants;
420     var values = links.GetLink(link);
421     var equalityComparer = EqualityComparer<TLink>.Default;
422     return equalityComparer.Equals(values[constants.SourcePart], source) &&
    ↳ equalityComparer.Equals(values[constants.TargetPart], target);
423 }
424
425 /// <summary>
426 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
427 /// </summary>
428 /// <param name="links">Хранилище связей.</param>
429 /// <param name="source">Индекс связи, которая является началом для искомой
    ↳ связи.</param>
430 /// <param name="target">Индекс связи, которая является концом для искомой
    ↳ связи.</param>
431 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
    ↳ (концом).</returns>
432 [MethodImpl(MethodImplOptions.AggressiveInlining)]
433 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink
    ↳ source, TLink target)
434 {
435     var contants = links.Constants;
436     var setter = new Setter<TLink, TLink>(contants.Continue, contants.Break,
    ↳ default);
437     links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
438     return setter.Result;
439 }
440
441 /// <param name="links">Хранилище связей.</param>
442 [MethodImpl(MethodImplOptions.AggressiveInlining)]
443 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
444 {
    var link = links.Create();

```

```

445     return links.Update(link, link, link);
446 }
447
448 /// <param name="links">Хранилище связей.</param>
449 [MethodImpl(MethodImplOptions.AggressiveInlining)]
450 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink
↳ source, TLink target) => links.Update(links.Create(), source, target);
451
452 /// <summary>
453 /// Обновляет связь с указанными началом (Source) и концом (Target)
454 /// на связь с указанными началом (NewSource) и концом (NewTarget).
455 /// </summary>
456 /// <param name="links">Хранилище связей.</param>
457 /// <param name="link">Индекс обновляемой связи.</param>
458 /// <param name="newSource">Индекс связи, которая является началом связи,
↳ на которую выполняется обновление.</param>
459 /// <param name="newTarget">Индекс связи, которая является концом связи, на
↳ которую выполняется обновление.</param>
460 /// <returns>Индекс обновлённой связи.</returns>
461 [MethodImpl(MethodImplOptions.AggressiveInlining)]
462 public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink
↳ newSource, TLink newTarget) => links.Update(new[] { link, newSource,
↳ newTarget });
463
464 /// <summary>
465 /// Обновляет связь с указанными началом (Source) и концом (Target)
466 /// на связь с указанными началом (NewSource) и концом (NewTarget).
467 /// </summary>
468 /// <param name="links">Хранилище связей.</param>
469 /// <param name="restrictions">Ограничения на содержимое связей. Каждое
↳ ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая
↳ ссылку на пустоту, Itself - требование установить ссылку на себя, 1..∞
↳ конкретный адрес другой связи.</param>
470 /// <returns>Индекс обновлённой связи.</returns>
471 [MethodImpl(MethodImplOptions.AggressiveInlining)]
472 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[]
↳ restrictions)
473 {
474     if (restrictions.Length == 2)
475     {
476         return links.Merge(restrictions[0], restrictions[1]);
477     }
478     if (restrictions.Length == 4)
479     {
480         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1], restrictions[2],
↳ restrictions[3]);
481     }
482     else
483     {
484         return links.Update(restrictions);
485     }
486 }
487
488 /// <summary>
489 /// Создаёт связь (если она не существовала), либо возвращает индекс
↳ существующей связи с указанными Source (началом) и Target (концом).
490 /// </summary>
491 /// <param name="links">Хранилище связей.</param>
492 /// <param name="source">Индекс связи, которая является началом на
↳ создаваемой связи.</param>

```

```

493 /// <param name="target">Индекс связи, которая является концом для
↳ создаваемой связи.</param>
494 /// <returns>Индекс связи, с указанным Source (началом) и Target
↳ (концом)</returns>
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source,
↳ TLink target)
497 {
498     var link = links.SearchOrDefault(source, target);
499     if (EqualityComparer<TLink>.Default.Equals(link, default))
500     {
501         link = links.CreateAndUpdate(source, target);
502     }
503     return link;
504 }
505
506 /// <summary>
507 /// Обновляет связь с указанными началом (Source) и концом (Target)
508 /// на связь с указанными началом (NewSource) и концом (NewTarget).
509 /// </summary>
510 /// <param name="links">Хранилище связей.</param>
511 /// <param name="source">Индекс связи, которая является началом
↳ обновляемой связи.</param>
512 /// <param name="target">Индекс связи, которая является концом обновляемой
↳ связи.</param>
513 /// <param name="newSource">Индекс связи, которая является началом связи,
↳ на которую выполняется обновление.</param>
514 /// <param name="newTarget">Индекс связи, которая является концом связи, на
↳ которую выполняется обновление.</param>
515 /// <returns>Индекс обновлённой связи.</returns>
516 [MethodImpl(MethodImplOptions.AggressiveInlining)]
517 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink
↳ source, TLink target, TLink newSource, TLink newTarget)
518 {
519     var equalityComparer = EqualityComparer<TLink>.Default;
520     var link = links.SearchOrDefault(source, target);
521     if (equalityComparer.Equals(link, default))
522     {
523         return links.CreateAndUpdate(newSource, newTarget);
524     }
525     if (equalityComparer.Equals(newSource, source) &&
↳ equalityComparer.Equals(newTarget, target))
526     {
527         return link;
528     }
529     return links.Update(link, newSource, newTarget);
530 }
531
532 /// <summary>Удаляет связь с указанными началом (Source) и концом
↳ (Target).</summary>
533 /// <param name="links">Хранилище связей.</param>
534 /// <param name="source">Индекс связи, которая является началом удаляемой
↳ связи.</param>
535 /// <param name="target">Индекс связи, которая является концом удаляемой
↳ связи.</param>
536 [MethodImpl(MethodImplOptions.AggressiveInlining)]
537 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source,
↳ TLink target)
538 {
539     var link = links.SearchOrDefault(source, target);
540     if (!EqualityComparer<TLink>.Default.Equals(link, default))

```



```

541     {
542         links.Delete(link);
543         return link;
544     }
545     return default;
546 }
547
548 /// <summary>Удаляет несколько связей.</summary>
549 /// <param name="links">Хранилище связей.</param>
550 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
551 [MethodImpl(MethodImplOptions.AggressiveInlining)]
552 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink>
    ↪ deletedLinks)
553 {
554     for (int i = 0; i < deletedLinks.Count; i++)
555     {
556         links.Delete(deletedLinks[i]);
557     }
558 }
559
560 // Replace one link with another (replaced link is deleted, children are updated or
    ↪ deleted)
561 public static TLink Merge<TLink>(this ILinks<TLink> links, TLink linkIndex,
    ↪ TLink newLink)
562 {
563     var equalityComparer = EqualityComparer<TLink>.Default;
564     if (equalityComparer.Equals(linkIndex, newLink))
565     {
566         return newLink;
567     }
568     var constants = links.Constants;
569     ulong referencesAsSourceCount = (Integer<TLink>)links.Count(constants.Any,
    ↪ linkIndex, constants.Any);
570     ulong referencesAsTargetCount = (Integer<TLink>)links.Count(constants.Any,
    ↪ constants.Any, linkIndex);
571     var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(linkIndex)) &&
    ↪ referencesAsSourceCount == 1 && referencesAsTargetCount == 1;
572     if (!isStandalonePoint)
573     {
574         var totalReferences = referencesAsSourceCount + referencesAsTargetCount;
575         if (totalReferences > 0)
576         {
577             var references = ArrayPool.Allocate<TLink>((long)totalReferences);
578             var referencesFiller = new ArrayFiller<TLink, TLink>(references,
    ↪ links.Constants.Continue);
579             links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any,
    ↪ linkIndex, constants.Any);
580             links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any,
    ↪ constants.Any, linkIndex);
581             for (ulong i = 0; i < referencesAsSourceCount; i++)
582             {
583                 var reference = references[i];
584                 if (equalityComparer.Equals(reference, linkIndex))
585                 {
586                     continue;
587                 }
588                 links.Update(reference, newLink, links.GetTarget(reference));
589             }
590             for (var i = (long)referencesAsSourceCount; i < references.Length; i++)
591             {
592                 var reference = references[i];

```

```

594         if (equalityComparer.Equals(reference, linkIndex))
595         {
596             continue;
597         }
598         links.Update(reference, links.GetSource(reference), newLink);
599     }
600     ArrayPool.Free(references);
601 }
602 }
603 }
604 links.Delete(linkIndex);
605 return newLink;
606 }
607 }
608 }

```

## ./Incrementers/FrequencyIncrementer.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Incrementers
5 {
6     public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>,
    ↪ Incrementer<TLink>
7     {
8         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪ EqualityComparer<TLink>.Default;
9
10         private readonly TLink _frequencyMarker;
11         private readonly TLink _unaryOne;
12         private readonly Incrementer<TLink> _unaryNumberIncrementer;
13
14         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink
    ↪ _unaryOne, Incrementer<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>,
    ↪ Incrementer<IList<TLink>>

```

```

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

```

#### ./Incrementers/UnaryNumberIncrementer.cs

```

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

```

```

25     }
26     else
27     {
28         return Links.GetOrCreate(source, Increment(target));
29     }
30 }
31 }
32 }

```

#### ./ISynchronizedLinks.cs

```

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

```

#### ./Link.cs

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using Platform.Exceptions;
5 using Platform.Ranges;
6 using Platform.Helpers.Singletons;
7 using Platform.Data.Constants;
8
9 namespace Platform.Data.Doublets
10 {
11     /// <summary>
12     /// Структура описывающая уникальную связь.
13     /// </summary>
14     public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>,
15     ↪ IList<TLink>
16     {
17         public static readonly Link<TLink> Null = new Link<TLink>();
18
19         private static readonly LinksCombinedConstants<bool, TLink, int> _constants =
20         ↪ Default<LinksCombinedConstants<bool, TLink, int>>.Instance;
21         private static readonly EqualityComparer<TLink> _equalityComparer =
22         ↪ EqualityComparer<TLink>.Default;
23
24         private const int Length = 3;
25
26         public readonly TLink Index;
27         public readonly TLink Source;
28         public readonly TLink Target;
29
30         public Link(params TLink[] values)
31         {
32             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
33             ↪ _constants.Null;
34             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
35             ↪ _constants.Null;
36             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
37             ↪ _constants.Null;
38         }
39
40         public Link(IList<TLink> values)
41         {
42         }
43     }
44 }

```

```

36     Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
    ↪     _constants.Null;
37     Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
    ↪     _constants.Null;
38     Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
    ↪     _constants.Null;
39 }
40
41 public Link(TLink index, TLink source, TLink target)
42 {
43     Index = index;
44     Source = source;
45     Target = target;
46 }
47
48 public Link(TLink source, TLink target)
49 : this(_constants.Null, source, target)
50 {
51     Source = source;
52     Target = target;
53 }
54
55 public static Link<TLink> Create(TLink source, TLink target) => new
    ↪     Link<TLink>(source, target);
56
57 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
58
59 public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
60     && _equalityComparer.Equals(Source, _constants.Null)
61     && _equalityComparer.Equals(Target, _constants.Null);
62
63 public override bool Equals(object other) => other is Link<TLink> &&
    ↪     Equals((Link<TLink>)other);
64
65 public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index,
    ↪     other.Index)
66     && _equalityComparer.Equals(Source, other.Source)
67     && _equalityComparer.Equals(Target, other.Target);
68
69 public static string ToString(TLink index, TLink source, TLink target) =>
    ↪     $"{index}: {source}->{target}";
70
71 public static string ToString(TLink source, TLink target) => $"{source}->{target}";
72
73 public static implicit operator TLink[(Link<TLink> link) => link.ToArray()];
74
75 public static implicit operator Link<TLink>(TLink[] linkArray) => new
    ↪     Link<TLink>(linkArray);
76
77 public TLink[] ToArray()
78 {
79     var array = new TLink[Length];
80     CopyTo(array, 0);
81     return array;
82 }
83
84 public override string ToString() => _equalityComparer.Equals(Index,
    ↪     _constants.Null) ? ToString(Source, Target) : ToString(Index, Source, Target);
85
86 #region IList
87
88 public int Count => Length;
89
90 public bool IsReadOnly => true;

```

```

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

IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

public IEnumerator<TLink> GetEnumerator()
{
    yield return Index;
    yield return Source;
    yield return Target;
}

public void Add(TLink item) => throw new NotSupportedException();

public void Clear() => throw new NotSupportedException();

public bool Contains(TLink item) => IndexOf(item) >= 0;

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

public bool Remove(TLink item) =>
    ↪     Throw.A.NotSupportedExceptionAndReturn<bool>();

public int IndexOf(TLink item)
{
    if (_equalityComparer.Equals(Index, item))
    {
        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
13     ⇨ Platform.Data.Doublets 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
18         public TLink GetValue(TLink @object, TLink property)
19         {
20             var objectProperty = Links.SearchOrDefault(@object, property);
21             if (_equalityComparer.Equals(objectProperty, default))
22             {
23                 return default;
24             }
25             var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
26             if (valueLink == null)
27             {
28                 return default;
29             }
30             var value = Links.GetTarget(valueLink[Links.Constants.IndexPart]);
31             return value;
32         }
33
34         public void SetValue(TLink @object, TLink property, TLink value)
35         {
36             var objectProperty = Links.GetOrCreate(@object, property);
37             Links.DeleteMany(Links.All(Links.Constants.Any, objectProperty).Select(link =>
38                 ⇨ link[Links.Constants.IndexPart]).ToList());
39             Links.GetOrCreate(objectProperty, value);
40         }
41     }
42 }

```

### ./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
16             ⇨ frequencyPropertyMarker, TLink frequencyMarker) : base(links)
17         {
18             _frequencyPropertyMarker = frequencyPropertyMarker;
19             _frequencyMarker = frequencyMarker;
20         }
21     }
22 }

```

```

17     }
18
19     public TLink Get(TLink link)
20     {
21         var property = Links.SearchOrDefault(link, _frequencyPropertyMarker);
22         var container = GetContainer(property);
23         var frequency = GetFrequency(container);
24         return frequency;
25     }
26
27     private TLink GetContainer(TLink property)
28     {
29         var frequencyContainer = default(TLink);
30         if (_equalityComparer.Equals(property, default))
31         {
32             return frequencyContainer;
33         }
34         Links.Each(candidate =>
35         {
36             var candidateTarget = Links.GetTarget(candidate);
37             var frequencyTarget = Links.GetTarget(candidateTarget);
38             if (_equalityComparer.Equals(frequencyTarget, _frequencyMarker))
39             {
40                 frequencyContainer = Links.GetIndex(candidate);
41                 return Links.Constants.Break;
42             }
43             return Links.Constants.Continue;
44         }, Links.Constants.Any, property, Links.Constants.Any);
45         return frequencyContainer;
46     }
47
48     private TLink GetFrequency(TLink container) =>
49     ↪ _equalityComparer.Equals(container, default) ? default :
50     ↪ Links.GetTarget(container);
51
52     public void Set(TLink link, TLink frequency)
53     {
54         var property = Links.GetOrCreate(link, _frequencyPropertyMarker);
55         var container = GetContainer(property);
56         if (_equalityComparer.Equals(container, default))
57         {
58             Links.GetOrCreate(property, frequency);
59         }
60         else
61         {
62             Links.Update(container, property, frequency);
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,
27     ↪ ILinks<TLink>
28     {
29         private static readonly EqualityComparer<TLink> _equalityComparer =
30         ↪ EqualityComparer<TLink>.Default;
31         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
32
33         /// <summary>Возвращает размер одной связи в байтах.</summary>
34         public static readonly int LinkSizeInBytes = StructureHelpers.SizeOf<Link>();
35
36         public static readonly int LinkHeaderSizeInBytes =
37         ↪ StructureHelpers.SizeOf<LinksHeader>();
38
39         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
40
41         private struct Link
42         {
43             public static readonly int SourceOffset = Marshal.OffsetOf(typeof(Link),
44             ↪ nameof(Source)).ToInt32();
45             public static readonly int TargetOffset = Marshal.OffsetOf(typeof(Link),
46             ↪ nameof(Target)).ToInt32();
47             public static readonly int LeftAsSourceOffset = Marshal.OffsetOf(typeof(Link),
48             ↪ nameof(LeftAsSource)).ToInt32();
49             public static readonly int RightAsSourceOffset = Marshal.OffsetOf(typeof(Link),
50             ↪ nameof(RightAsSource)).ToInt32();
51             public static readonly int SizeAsSourceOffset = Marshal.OffsetOf(typeof(Link),
52             ↪ nameof(SizeAsSource)).ToInt32();
53             public static readonly int LeftAsTargetOffset = Marshal.OffsetOf(typeof(Link),
54             ↪ nameof(LeftAsTarget)).ToInt32();
55             public static readonly int RightAsTargetOffset = Marshal.OffsetOf(typeof(Link),
56             ↪ nameof(RightAsTarget)).ToInt32();
57             public static readonly int SizeAsTargetOffset = Marshal.OffsetOf(typeof(Link),
58             ↪ nameof(SizeAsTarget)).ToInt32();
59
60             public TLink Source;
61             public TLink Target;
62             public TLink LeftAsSource;
63             public TLink RightAsSource;
64             public TLink SizeAsSource;
65             public TLink LeftAsTarget;
66             public TLink RightAsTarget;
67             public TLink SizeAsTarget;
68
69             [MethodImpl(MethodImplOptions.AggressiveInlining)]
70             public static TLink GetSource(IntPtr pointer) => (pointer +
71             ↪ SourceOffset).GetValue<TLink>();
72             [MethodImpl(MethodImplOptions.AggressiveInlining)]
73             public static TLink GetTarget(IntPtr pointer) => (pointer +
74             ↪ TargetOffset).GetValue<TLink>();
75         }
76     }
77 }

```

```

62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 public static TLink GetLeftAsSource(IntPtr pointer) => (pointer +
    ↳ LeftAsSourceOffset).GetValue<TLink>();
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public static TLink GetRightAsSource(IntPtr pointer) => (pointer +
    ↳ RightAsSourceOffset).GetValue<TLink>();
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 public static TLink GetSizeAsSource(IntPtr pointer) => (pointer +
    ↳ SizeAsSourceOffset).GetValue<TLink>();
68 [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 +
    ↳ FirstAsSourceOffset;
129 [MethodImpl(MethodImplOptions.AggressiveInlining)]
130 public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
    ↳ FirstAsTargetOffset;
131
132 [MethodImpl(MethodImplOptions.AggressiveInlining)]
133 public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
    ↳ AllocatedLinksOffset).SetValue(value);
134 [MethodImpl(MethodImplOptions.AggressiveInlining)]
135 public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
    ↳ ReservedLinksOffset).SetValue(value);
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
    ↳ FreeLinksOffset).SetValue(value);
138 [MethodImpl(MethodImplOptions.AggressiveInlining)]
139 public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
    ↳ FirstFreeLinkOffset).SetValue(value);
140 [MethodImpl(MethodImplOptions.AggressiveInlining)]
141 public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
    ↳ FirstAsSourceOffset).SetValue(value);
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
    ↳ FirstAsTargetOffset).SetValue(value);
144 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

```

198     _memory.UsedCapacity =
    ↪ ((long)(Integer<TLink>)LinksHeader.GetAllocatedLinks(_header) *
    ↪ LinkSizeInBytes) + LinkHeaderSizeInBytes;
199     // Гарантия корректности _header->ReservedLinks относительно
    ↪ _memory.ReservedCapacity
200     LinksHeader.SetReservedLinks(_header,
    ↪ (Integer<TLink>)((_memory.ReservedCapacity - LinkHeaderSizeInBytes) /
    ↪ LinkSizeInBytes));
201 }
202
203 [MethodImpl(MethodImplOptions.AggressiveInlining)]
204 public TLink Count(ICollection<TLink> restrictions)
205 {
206     // Если нет ограничений, тогда возвращаем общее число связей находящихся в
    ↪ хранилище.
207     if (restrictions.Count == 0)
208     {
209         return Total;
210     }
211     if (restrictions.Count == 1)
212     {
213         var index = restrictions[Constants.IndexPart];
214         if (_equalityComparer.Equals(index, Constants.Any))
215         {
216             return Total;
217         }
218         return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
219     }
220     if (restrictions.Count == 2)
221     {
222         var index = restrictions[Constants.IndexPart];
223         var value = restrictions[1];
224         if (_equalityComparer.Equals(index, Constants.Any))
225         {
226             if (_equalityComparer.Equals(value, Constants.Any))
227             {
228                 return Total; // Any - как отсутствие ограничения
229             }
230             return Add(_sourcesTreeMethods.CalculateReferences(value),
    ↪ _targetsTreeMethods.CalculateReferences(value));
231         }
232         else
233         {
234             if (!Exists(index))
235             {
236                 return Integer<TLink>.Zero;
237             }
238             if (_equalityComparer.Equals(value, Constants.Any))
239             {
240                 return Integer<TLink>.One;
241             }
242             var storedLinkValue = GetLinkUnsafe(index);
243             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
244                 _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
245             {
246                 return Integer<TLink>.One;
247             }
248             return Integer<TLink>.Zero;
249         }
250     }
251     if (restrictions.Count == 3)
252     {

```

```

253 var index = restrictions[Constants.IndexPart];
254 var source = restrictions[Constants.SourcePart];
255 var target = restrictions[Constants.TargetPart];
256
257 if (_equalityComparer.Equals(index, Constants.Any))
258 {
259     if (_equalityComparer.Equals(source, Constants.Any) &&
        ↪ _equalityComparer.Equals(target, Constants.Any))
260     {
261         return Total;
262     }
263     else if (_equalityComparer.Equals(source, Constants.Any))
264     {
265         return _targetsTreeMethods.CalculateReferences(target);
266     }
267     else if (_equalityComparer.Equals(target, Constants.Any))
268     {
269         return _sourcesTreeMethods.CalculateReferences(source);
270     }
271     else //if(source != Any && target != Any)
272     {
273         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
274         var link = _sourcesTreeMethods.Search(source, target);
275         return _equalityComparer.Equals(link, Constants.Null) ?
            ↪ Integer<TLink>.Zero : Integer<TLink>.One;
276     }
277 }
278 else
279 {
280     if (!Exists(index))
281     {
282         return Integer<TLink>.Zero;
283     }
284     if (_equalityComparer.Equals(source, Constants.Any) &&
        ↪ _equalityComparer.Equals(target, Constants.Any))
285     {
286         return Integer<TLink>.One;
287     }
288     var storedLinkValue = GetLinkUnsafe(index);
289     if (!_equalityComparer.Equals(source, Constants.Any) &&
        ↪ !_equalityComparer.Equals(target, Constants.Any))
290     {
291         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source)
            ↪ &&
292             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
293         {
294             return Integer<TLink>.One;
295         }
296         return Integer<TLink>.Zero;
297     }
298     var value = default(TLink);
299     if (_equalityComparer.Equals(source, Constants.Any))
300     {
301         value = target;
302     }
303     if (_equalityComparer.Equals(target, Constants.Any))
304     {
305         value = source;
306     }
307     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
308         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
309     {

```

```

310         return Integer<TLink>.One;
311     }
312     return Integer<TLink>.Zero;
313 }
314 }
315 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↪ поддерживаются.");
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
{
    if (restrictions.Count == 0)
    {
        for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
            ↪ (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
            ↪ Increment(link))
        {
            if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
                ↪ Constants.Break))
            {
                return Constants.Break;
            }
        }
        return Constants.Continue;
    }
    if (restrictions.Count == 1)
    {
        var index = restrictions[Constants.IndexPart];
        if (_equalityComparer.Equals(index, Constants.Any))
        {
            return Each(handler, ArrayPool<TLink>.Empty);
        }
        if (!Exists(index))
        {
            return Constants.Continue;
        }
        return handler(GetLinkStruct(index));
    }
    if (restrictions.Count == 2)
    {
        var index = restrictions[Constants.IndexPart];
        var value = restrictions[1];
        if (_equalityComparer.Equals(index, Constants.Any))
        {
            if (_equalityComparer.Equals(value, Constants.Any))
            {
                return Each(handler, ArrayPool<TLink>.Empty);
            }
            if (_equalityComparer.Equals(Each(handler, new[] { index, value,
                ↪ Constants.Any }), Constants.Break))
            {
                return Constants.Break;
            }
            return Each(handler, new[] { index, Constants.Any, value });
        }
        else
        {
            if (!Exists(index))
            {

```



```

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) ?
405                 ↪ Constants.Continue : 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                 ↪ &&
425                 _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
426             {

```

```

422         return handler(GetLinkStruct(index));
423     }
424     return Constants.Continue;
425 }
426 var value = default(TLink);
427 if (_equalityComparer.Equals(source, Constants.Any))
428 {
429     value = target;
430 }
431 if (_equalityComparer.Equals(target, Constants.Any))
432 {
433     value = source;
434 }
435 if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
436     _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
437 {
438     return handler(GetLinkStruct(index));
439 }
440 return Constants.Continue;
441 }
442 }
443 throw new NotSupportedException("Другие размеры и способы ограничений не
444     ↪ поддерживаются.");
445 }
446
447 /// <remarks>
448 /// TODO: Возможно можно перемещать значения, если указан индекс, но
449     ↪ значение существует в другом месте (но не в менеджере памяти, а в логике
450     ↪ Links)
451 /// </remarks>
452 [MethodImpl(MethodImplOptions.AggressiveInlining)]
453 public TLink Update(IList<TLink> values)
454 {
455     var linkIndex = values[Constants.IndexPart];
456     var link = GetLinkUnsafe(linkIndex);
457     // Будет корректно работать только в том случае, если пространство
458     ↪ выделенной связи предварительно заполнено нулями
459     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
460     {
461         _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
462             ↪ linkIndex);
463     }
464     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
465     {
466         _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
467             ↪ linkIndex);
468     }
469     Link.SetSource(link, values[Constants.SourcePart]);
470     Link.SetTarget(link, values[Constants.TargetPart]);
471     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
472     {
473         _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer(_header),
474             ↪ linkIndex);
475     }
476     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
477     {
478         _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTargetPointer(_header),
479             ↪ linkIndex);
480     }
481     return linkIndex;
482 }

```

```

475 [MethodImpl(MethodImplOptions.AggressiveInlining)]
476 public Link<TLink> GetLinkStruct(TLink linkIndex)
477 {
478     var link = GetLinkUnsafe(linkIndex);
479     return new Link<TLink>(linkIndex, Link.GetSource(link), Link.GetTarget(link));
480 }
481
482 [MethodImpl(MethodImplOptions.AggressiveInlining)]
483 private IntPtr GetLinkUnsafe(TLink linkIndex) =>
484     ↪ _links.GetElement(LinkSizeInBytes, linkIndex);
485
486 /// <remarks>
487 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не
488     ↪ заполняет пространство
489 /// </remarks>
490 public TLink Create()
491 {
492     var freeLink = LinksHeader.GetFirstFreeLink(_header);
493     if (!equalityComparer.Equals(freeLink, Constants.Null))
494     {
495         _unusedLinksListMethods.Detach(freeLink);
496     }
497     else
498     {
499         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
500             ↪ Constants.MaxPossibleIndex) > 0)
501         {
502             throw new LinksLimitReachedException((Integer<TLink>)Constants.MaxP
503                 ↪ ossibleIndex);
504         }
505         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
506             ↪ Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
507         {
508             memory.ReservedCapacity += _memory.ReservationStep;
509             SetPointers(_memory);
510             LinksHeader.SetReservedLinks(_header,
511                 ↪ (Integer<TLink>)(_memory.ReservedCapacity / LinkSizeInBytes));
512         }
513         LinksHeader.SetAllocatedLinks(_header,
514             ↪ Increment(LinksHeader.GetAllocatedLinks(_header)));
515         memory.UsedCapacity += LinkSizeInBytes;
516         freeLink = LinksHeader.GetAllocatedLinks(_header);
517     }
518     return freeLink;
519 }
520
521 public void Delete(TLink link)
522 {
523     if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)
524     {
525         _unusedLinksListMethods.AttachAsFirst(link);
526     }
527     else if (_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
528     {
529         LinksHeader.SetAllocatedLinks(_header,
530             ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
531         memory.UsedCapacity -= LinkSizeInBytes;
532         // Убираем все связи, находящиеся в списке свободных в конце файла, до
533             ↪ тех пор, пока не дойдём до первой существующей связи
534         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)

```

```

527 while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
528     ↪ Integer<TLink>.Zero) > 0) &&
529     ↪ IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
530 {
531     unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
532     LinksHeader.SetAllocatedLinks(_header,
533         ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
534     _memory.UsedCapacity -= LinkSizeInBytes;
535 }
536 }
537
538 /// <remarks>
539 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том
540     ↪ случае, если адрес реально поменялся
541
542 /// Указатель this.links может быть в том же месте,
543     ↪ так как 0-я связь не используется и имеет такой же размер как Header,
544     ↪ поэтому header размещается в том же месте, что и 0-я связь
545 /// </remarks>
546 private void SetPointers(IDirectMemory memory)
547 {
548     if (memory == null)
549     {
550         links = IntPtr.Zero;
551         _header = links;
552         _unusedLinksListMethods = null;
553         _targetsTreeMethods = null;
554         _unusedLinksListMethods = null;
555     }
556     else
557     {
558         links = memory.Pointer;
559         _header = links;
560         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
561         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
562         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
563     }
564 }
565
566 [MethodImpl(MethodImplOptions.AggressiveInlining)]
567 private bool Exists(TLink link)
568     => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
569     && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
570     && !IsUnusedLink(link);
571
572 [MethodImpl(MethodImplOptions.AggressiveInlining)]
573 private bool IsUnusedLink(TLink link)
574     => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
575     || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)),
576         ↪ Constants.Null)
577         && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)),
578             ↪ Constants.Null));
579
580 #region DisposableBase
581
582 protected override bool AllowMultipleDisposeCalls => true;
583
584 protected override void DisposeCore(bool manual, bool wasDisposed)
585 {
586     if (!wasDisposed)
587     {
588         SetPointers(null);

```

```

584     }
585     Disposable.TryDispose(_memory);
586 }
587
588 #endregion
589 }
590 }

```

# ./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) =>
31                 ↳ (_links.GetElement(LinkSizeInBytes, element) +
32                 ↳ Link.TargetOffset).GetValue<TLink>();
33
34             protected override TLink GetSize() => (_header +
35                 ↳ LinksHeader.FreeLinksOffset).GetValue<TLink>();
36
37             protected override void SetFirst(TLink element) => (_header +
38                 ↳ LinksHeader.FirstFreeLinkOffset).SetValue(element);
39
40             protected override void SetLast(TLink element) => (_header +
41                 ↳ LinksHeader.LastFreeLinkOffset).SetValue(element);
42
43             protected override void SetPrevious(TLink element, TLink previous) =>
44                 ↳ (_links.GetElement(LinkSizeInBytes, element) +
45                 ↳ Link.SourceOffset).SetValue(previous);
46
47             protected override void SetNext(TLink element, TLink next) =>
48                 ↳ (_links.GetElement(LinkSizeInBytes, element) +
49                 ↳ Link.TargetOffset).SetValue(next);
50
51             protected override void SetSize(TLink size) => (_header +
52                 ↳ LinksHeader.FreeLinksOffset).SetValue(size);
53         }
54     }
55 }

```

```

41 }

```

# ./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Numbers;
6  using Platform.Unsafe;
7  using Platform.Collections.Methods.Trees;
8  using Platform.Data.Constants;
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory
11 {
12     partial class ResizableDirectMemoryLinks<TLink>
13     {
14         private abstract class LinksTreeMethodsBase :
15             ↳ SizedAndThreadedAVLBalancedTreeMethods<TLink>
16         {
17             private readonly ResizableDirectMemoryLinks<TLink> _memory;
18             private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
19             protected readonly IntPtr Links;
20             protected readonly IntPtr Header;
21
22             protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
23             {
24                 Links = memory._links;
25                 Header = memory._header;
26                 _memory = memory;
27                 _constants = memory.Constants;
28             }
29
30             [MethodImpl(MethodImplOptions.AggressiveInlining)]
31             protected abstract TLink GetTreeRoot();
32
33             [MethodImpl(MethodImplOptions.AggressiveInlining)]
34             protected abstract TLink GetBasePartValue(TLink link);
35
36             public TLink this[TLink index]
37             {
38                 get
39                 {
40                     var root = GetTreeRoot();
41                     if (GreaterOrEqualThan(index, GetSize(root)))
42                     {
43                         return GetZero();
44                     }
45                     while (!EqualToZero(root))
46                     {
47                         var left = GetLeftOrDefault(root);
48                         var leftSize = GetSizeOrZero(left);
49                         if (LessThan(index, leftSize))
50                         {
51                             root = left;
52                             continue;
53                         }
54                     }
55                     if (IsEquals(index, leftSize))
56                     {
57                         return root;
58                     }
59                     root = GetRightOrDefault(root);
60                     index = Subtract(index, Increment(leftSize));
61                 }
62             }
63         }
64     }
65 }

```

```

60         return GetZero(); // TODO: Impossible situation exception (only if tree
        ↪ 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         }
96         root = GetRightOrDefault(root);
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     }

```

```

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     }

```

```

174     }
175
176     protected override void SetLeft(TLink node, TLink left) =>
177     ↪ (Links.GetElement(LinkSizeInBytes, node) +
178     ↪ Link.LeftAsSourceOffset).SetValue(left);
179
180     protected override void SetRight(TLink node, TLink right) =>
181     ↪ (Links.GetElement(LinkSizeInBytes, node) +
182     ↪ Link.RightAsSourceOffset).SetValue(right);
183
184     protected override void SetSize(TLink node, TLink size)
185     {
186         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
187         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
188         (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsSourceOffset).SetValue(
189         ↪ e(BitwiseHelpers.PartialWrite(previousValue, size, 5,
190         ↪ -5)));
191     }
192
193     protected override bool GetLeftIsChild(TLink node)
194     {
195         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
196         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
197         return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 4, 1);
198     }
199
200     protected override void SetLeftIsChild(TLink node, bool value)
201     {
202         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
203         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
204         var modified = BitwiseHelpers.PartialWrite(previousValue,
205         ↪ (TLink)(Integer<TLink>)value, 4, 1);
206         (Links.GetElement(LinkSizeInBytes, node) +
207         ↪ Link.SizeAsSourceOffset).SetValue(modified);
208     }
209
210     protected override bool GetRightIsChild(TLink node)
211     {
212         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
213         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
214         return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 3, 1);
215     }
216
217     protected override void SetRightIsChild(TLink node, bool value)
218     {
219         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
220         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
221         var modified = BitwiseHelpers.PartialWrite(previousValue,
222         ↪ (TLink)(Integer<TLink>)value, 3, 1);
223         (Links.GetElement(LinkSizeInBytes, node) +
224         ↪ Link.SizeAsSourceOffset).SetValue(modified);
225     }
226
227     protected override sbyte GetBalance(TLink node)
228     {
229         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
230         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
231         var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue,
232         ↪ 0, 3);
233         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
234         ↪ 124 : value & 3);

```

```

217         return unpackedValue;
218     }
219
220     protected override void SetBalance(TLink node, sbyte value)
221     {
222         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
223         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
224         var packagedValue = (TLink)(Integer<TLink>)(((byte)value >> 5) & 4) |
225         ↪ value & 3);
226         var modified = BitwiseHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
227         (Links.GetElement(LinkSizeInBytes, node) +
228         ↪ Link.SizeAsSourceOffset).SetValue(modified);
229     }
230
231     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
232     {
233         var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
234         ↪ Link.SourceOffset).GetValue<TLink>();
235         var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
236         ↪ Link.SourceOffset).GetValue<TLink>();
237         return LessThan(firstSource, secondSource) ||
238         ↪ (IsEquals(firstSource, secondSource) &&
239         ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
240         ↪ Link.TargetOffset).GetValue<TLink>(),
241         ↪ (Links.GetElement(LinkSizeInBytes, second) +
242         ↪ Link.TargetOffset).GetValue<TLink>()));
243     }
244
245     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
246     {
247         var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
248         ↪ Link.SourceOffset).GetValue<TLink>();
249         var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
250         ↪ Link.SourceOffset).GetValue<TLink>();
251         return GreaterThan(firstSource, secondSource) ||
252         ↪ (IsEquals(firstSource, secondSource) &&
253         ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
254         ↪ Link.TargetOffset).GetValue<TLink>(),
255         ↪ (Links.GetElement(LinkSizeInBytes, second) +
256         ↪ Link.TargetOffset).GetValue<TLink>()));
257     }
258
259     protected override TLink GetTreeRoot() => (Header +
260     ↪ LinksHeader.FirstAsSourceOffset).GetValue<TLink>();
261
262     protected override TLink GetBasePartValue(TLink link) =>
263     ↪ (Links.GetElement(LinkSizeInBytes, link) +
264     ↪ Link.SourceOffset).GetValue<TLink>();
265
266     /// <summary>
267     /// Выполняет поиск и возвращает индекс связи с указанными Source
268     ↪ (началом) и Target (концом)
269     /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
270     /// </summary>
271     /// <param name="source">Индекс связи, которая является началом на
272     ↪ искомой связи.</param>
273     /// <param name="target">Индекс связи, которая является концом на искомой
274     ↪ связи.</param>
275     /// <returns>Индекс искомой связи.</returns>

```

```

255 public TLink Search(TLink source, TLink target)
256 {
257     var root = GetTreeRoot();
258     while (!EqualToZero(root))
259     {
260         var rootSource = (Links.GetElement(LinkSizeInBytes, root) +
261             ↪ Link.SourceOffset).GetValue<TLink>();
262         var rootTarget = (Links.GetElement(LinkSizeInBytes, root) +
263             ↪ Link.TargetOffset).GetValue<TLink>();
264         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
265             ↪ node.Key < root.Key
266         {
267             root = GetLeftOrDefault(root);
268         }
269         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
270             ↪ // node.Key > root.Key
271         {
272             root = GetRightOrDefault(root);
273         }
274         else // node.Key == root.Key
275         {
276             return root;
277         }
278     }
279     return GetZero();
280 }
281 [MethodImpl(MethodImplOptions.AggressiveInlining)]
282 private bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget, TLink
283     ↪ secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
284     ↪ (IsEquals(firstSource, secondSource) && LessThan(firstTarget,
285     ↪ secondTarget));
286 }
287 [MethodImpl(MethodImplOptions.AggressiveInlining)]
288 private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
289     ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource,
290     ↪ secondSource) || (IsEquals(firstSource, secondSource) &&
291     ↪ GreaterThan(firstTarget, secondTarget));
292 }
293 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
294 {
295     public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
296     : base(memory)
297     {
298     }
299     protected override IntPtr GetLeftPointer(TLink node) =>
300     ↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset;
301     protected override IntPtr GetRightPointer(TLink node) =>
302     ↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset;
303     protected override TLink GetLeftValue(TLink node) =>
304     ↪ (Links.GetElement(LinkSizeInBytes, node) +
305     ↪ Link.LeftAsTargetOffset).GetValue<TLink>();
306     protected override TLink GetRightValue(TLink node) =>
307     ↪ (Links.GetElement(LinkSizeInBytes, node) +
308     ↪ Link.RightAsTargetOffset).GetValue<TLink>();

```

```

300 protected override TLink GetSize(TLink node)
301 {
302     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
303     ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
304     return BitwiseHelpers.PartialRead(previousValue, 5, -5);
305 }
306 protected override void SetLeft(TLink node, TLink left) =>
307     ↪ (Links.GetElement(LinkSizeInBytes, node) +
308     ↪ Link.LeftAsTargetOffset).SetValue(left);
309 protected override void SetRight(TLink node, TLink right) =>
310     ↪ (Links.GetElement(LinkSizeInBytes, node) +
311     ↪ Link.RightAsTargetOffset).SetValue(right);
312 protected override void SetSize(TLink node, TLink size)
313 {
314     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
315     ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
316     (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsTargetOffset).SetValu
317     ↪ e(BitwiseHelpers.PartialWrite(previousValue, size, 5,
318     ↪ -5));
319 }
320 protected override bool GetLeftIsChild(TLink node)
321 {
322     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
323     ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
324     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 4, 1);
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 protected override bool GetRightIsChild(TLink node)
336 {
337     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
338     ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
339     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 3, 1);
340 }
341 protected override void SetRightIsChild(TLink node, bool value)
342 {
343     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
344     ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
345     var modified = BitwiseHelpers.PartialWrite(previousValue,
346     ↪ (TLink)(Integer<TLink>)value, 3, 1);
347     (Links.GetElement(LinkSizeInBytes, node) +
348     ↪ Link.SizeAsTargetOffset).SetValue(modified);
349 }
350 protected override sbyte GetBalance(TLink node)
351 {

```

```

344     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
345     var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue,
    ↪ 0, 3);
346     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
    ↪ 124 : value & 3);
347     return unpackedValue;
348 }
349
350 protected override void SetBalance(TLink node, sbyte value)
351 {
352     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
353     var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) |
    ↪ value & 3);
354     var modified = BitwiseHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
355     (Links.GetElement(LinkSizeInBytes, node) +
    ↪ Link.SizeAsTargetOffset).SetValue(modified);
356 }
357
358 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
359 {
360     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
    ↪ Link.TargetOffset).GetValue<TLink>();
361     var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
    ↪ Link.TargetOffset).GetValue<TLink>();
362     return LessThan(firstTarget, secondTarget) ||
363         (IsEquals(firstTarget, secondTarget) &&
    ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
    ↪ Link.SourceOffset).GetValue<TLink>(),
    ↪ (Links.GetElement(LinkSizeInBytes, second) +
    ↪ Link.SourceOffset).GetValue<TLink>()));
364 }
365
366 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
367 {
368     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
    ↪ Link.TargetOffset).GetValue<TLink>();
369     var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
    ↪ Link.TargetOffset).GetValue<TLink>();
370     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 использовать не список а дерево, так как так можно
↪ быстрее проверить на наличие связи внутри
67 private UnusedLinksListMethods _unusedLinksListMethods;
68
69 /// <summary>
70 /// Возвращает общее число связей находящихся в хранилище.
71 /// </summary>
72 private id Total => _header->AllocatedLinks - _header->FreeLinks;
73
74 // TODO: Дать возможность переопределять в конструкторе
75 public LinksCombinedConstants<id, id, int> Constants { get; }
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 ? 0UL : 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 :
            ↪ handler(GetLinkStruct(link));
307     }
308 }
309 else
310 {
311     if (!Exists(index))
312     {
313         return Constants.Continue;
314     }
315     if (source == Constants.Any && target == Constants.Any)
316     {
317         return handler(GetLinkStruct(index));
318     }
319     var storedLinkValue = GetLinkUnsafe(index);
320     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 пространство
407 /// </remarks>
408 public id Create()
409 {

```

```

410 var freeLink = _header->FirstFreeLink;
411 if (freeLink != Constants.Null)
412 {
413     _unusedLinksListMethods.Detach(freeLink);
414 }
415 else
416 {
417     if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
418     {
419         throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
420     }
421     if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
422     {
423         memory.ReservedCapacity += _memoryReservationStep;
424         SetPointers(_memory);
425         _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
426     }
427     _header->AllocatedLinks++;
428     memory.UsedCapacity += sizeof(Link);
429     freeLink = _header->AllocatedLinks;
430 }
431 return freeLink;
432 }
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
445         ↪ тех пор, пока не дойдём до первой существующей связи
446         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
447         while (_header->AllocatedLinks > 0 &&
448             ↪ IsUnusedLink(_header->AllocatedLinks))
449         {
450             _unusedLinksListMethods.Detach(_header->AllocatedLinks);
451             _header->AllocatedLinks--;
452             memory.UsedCapacity -= sizeof(Link);
453         }
454     }
455 }
456
457 /// <remarks>
458 /// TODO: Возможно это должно быть событием, вызываемым из IМemory, в том
459 ↪ случае, если адрес реально поменялся
460
461 /// Указатель this.links может быть в том же месте,
462 /// так как 0-я связь не используется и имеет такой же размер как Header,
463 /// поэтому header размещается в том же месте, что и 0-я связь
464 /// </remarks>
465 private void SetPointers(IResizableDirectMemory memory)
466 {
467     if (memory == null)
468     {
469         _header = null;
470         _links = null;
471         _unusedLinksListMethods = null;
472         _targetsTreeMethods = null;

```

```

470     _unusedLinksListMethods = null;
471 }
472 else
473 {
474     _header = (LinksHeader*)(void*)memory.Pointer;
475     _links = (Link*)(void*)memory.Pointer;
476     _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
477     _targetsTreeMethods = new LinksTargetsTreeMethods(this);
478     _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
479 }
480 }
481
482 [MethodImpl(MethodImplOptions.AggressiveInlining)]
483 private bool Exists(id link) => link >= Constants.MinPossibleIndex && link <=
484     ↪ _header->AllocatedLinks && !IsUnusedLink(link);
485
486 [MethodImpl(MethodImplOptions.AggressiveInlining)]
487 private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
488     || (_links[link].SizeAsSource == Constants.Null &&
489         ↪ _links[link].Source != Constants.Null);
490
491 #region Disposable
492
493 protected override bool AllowMultipleDisposeCalls => true;
494
495 protected override void DisposeCore(bool manual, bool wasDisposed)
496 {
497     if (!wasDisposed)
498     {
499         SetPointers(null);
500         Disposable.TryDispose(_memory);
501     }
502 }
503
504 #endregion
505 }

```

## ./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         protected override ulong GetSize() => _header->FreeLinks;
26
27         protected override void SetFirst(ulong element) => _header->FirstFreeLink =
28             ↳ element;
29
30         protected override void SetLast(ulong element) => _header->LastFreeLink =
31             ↳ element;
32
33         protected override void SetPrevious(ulong element, ulong previous) =>
34             ↳ _links[element].Source = previous;
35
36         protected override void SetNext(ulong element, ulong next) =>
37             ↳ _links[element].Target = next;
38
39         protected override void SetSize(ulong size) => _header->FreeLinks = size;
40     }
41 }
42

```

# ./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using System.Text;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Data.Constants;
7
8  namespace Platform.Data.Doublets.ResizableDirectMemory
9  {
10     unsafe partial class UInt64ResizableDirectMemoryLinks
11     {
12         private abstract class LinksTreeMethodsBase :
13             ↳ SizedAndThreadedAVLBalancedTreeMethods<ulong>
14         {
15             private readonly UInt64ResizableDirectMemoryLinks _memory;
16             private readonly LinksCombinedConstants<ulong, ulong, int> _constants;
17             protected readonly Link* Links;
18             protected readonly LinksHeader* Header;
19
20             protected LinksTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory)
21             {
22                 Links = memory._links;
23                 Header = memory._header;
24                 _memory = memory;
25                 _constants = memory.Constants;
26             }
27
28             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29             protected abstract ulong GetTreeRoot();
30
31             [MethodImpl(MethodImplOptions.AggressiveInlining)]
32             protected abstract ulong GetBasePartValue(ulong link);
33
34             public ulong this[ulong index]
35             {
36                 get
37                 {
38                     var root = GetTreeRoot();
39                     if (index >= GetSize(root))
40                     {
41                         return 0;
42                     }
43                     while (root != 0)

```

```

43         {
44             var left = GetLeftOrDefault(root);
45             var leftSize = GetSizeOrZero(left);
46             if (index < leftSize)
47             {
48                 root = left;
49                 continue;
50             }
51             if (index == leftSize)
52             {
53                 return root;
54             }
55             root = GetRightOrDefault(root);
56             index -= leftSize + 1;
57         }
58         return 0; // TODO: Impossible situation exception (only if tree structure
59             ↳ broken)
60     }
61 }
62
63 // TODO: Return indices range instead of references count
64 public ulong CalculateReferences(ulong link)
65 {
66     var root = GetTreeRoot();
67     var total = GetSize(root);
68     var totalRightIgnore = 0UL;
69     while (root != 0)
70     {
71         var @base = GetBasePartValue(root);
72         if (@base <= link)
73         {
74             root = GetRightOrDefault(root);
75         }
76         else
77         {
78             totalRightIgnore += GetRightSize(root) + 1;
79             root = GetLeftOrDefault(root);
80         }
81     }
82     root = GetTreeRoot();
83     var totalLeftIgnore = 0UL;
84     while (root != 0)
85     {
86         var @base = GetBasePartValue(root);
87         if (@base >= link)
88         {
89             root = GetLeftOrDefault(root);
90         }
91         else
92         {
93             totalLeftIgnore += GetLeftSize(root) + 1;
94             root = GetRightOrDefault(root);
95         }
96     }
97     return total - totalRightIgnore - totalLeftIgnore;
98 }
99
100 public ulong EachReference(ulong link, Func<IList<ulong>, ulong> handler)
101 {
102     var root = GetTreeRoot();
103     if (root == 0)
104     {

```

```

104         return _constants.Continue;
105     }
106     ulong first = 0, current = root;
107     while (current != 0)
108     {
109         var @base = GetBasePartValue(current);
110         if (@base >= link)
111         {
112             if (@base == link)
113             {
114                 first = current;
115             }
116             current = GetLeftOrDefault(current);
117         }
118         else
119         {
120             current = GetRightOrDefault(current);
121         }
122     }
123     if (first != 0)
124     {
125         current = first;
126         while (true)
127         {
128             if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
129             {
130                 return _constants.Break;
131             }
132             current = GetNext(current);
133             if (current == 0 || GetBasePartValue(current) != link)
134             {
135                 break;
136             }
137         }
138     }
139     return _constants.Continue;
140 }
141
142 protected override void PrintNodeValue(ulong node, StringBuilder sb)
143 {
144     sb.Append(' ');
145     sb.Append(Links[node].Source);
146     sb.Append('-');
147     sb.Append('>');
148     sb.Append(Links[node].Target);
149 }
150
151 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
152 {
153     public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)
154         : base(memory)
155     {
156     }
157 }
158
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;

```

```

165 protected override ulong GetRightValue(ulong node) => Links[node].RightAsSource;
166
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) =>
175     ↳ Links[node].LeftAsSource = left;
176
177 protected override void SetRight(ulong node, ulong right) =>
178     ↳ Links[node].RightAsSource = 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,
199     ↳ (ulong)(Integer)value, 4, 1);
200     var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
201     Links[node].SizeAsSource = modified;
202 }
203
204 protected override bool GetRightIsChild(ulong node)
205 {
206     var previousValue = Links[node].SizeAsSource;
207     //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
208     return (previousValue & 8) >> 3 == 1UL;
209 }
210
211 protected override void SetRightIsChild(ulong node, bool value)
212 {
213     var previousValue = Links[node].SizeAsSource;
214     //var modified = MathHelpers.PartialWrite(previousValue,
215     ↳ (ulong)(Integer)value, 3, 1);
216     var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
217     Links[node].SizeAsSource = modified;
218 }
219
220 protected override sbyte GetBalance(ulong node)
221 {
222     var previousValue = Links[node].SizeAsSource;
223     //var value = MathHelpers.PartialRead(previousValue, 0, 3);
224     var value = previousValue & 7;
225     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
226     ↳ 124 : value & 3);

```

```

222     return unpackedValue;
223 }
224
225 protected override void SetBalance(ulong node, sbyte value)
226 {
227     var previousValue = Links[node].SizeAsSource;
228     var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
229     //var modified = MathHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
230     var modified = (previousValue & 4294967288) | (packagedValue & 7);
231     Links[node].SizeAsSource = modified;
232 }
233
234 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
235     => Links[first].Source < Links[second].Source ||
236     (Links[first].Source == Links[second].Source && Links[first].Target <
237         ↳ Links[second].Target);
238
239 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
240     => Links[first].Source > Links[second].Source ||
241     (Links[first].Source == Links[second].Source && Links[first].Target >
242         ↳ Links[second].Target);
243
244 protected override ulong GetTreeRoot() => Header->FirstAsSource;
245
246 protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
247
248 /// <summary>
249 /// Выполняет поиск и возвращает индекс связи с указанными Source
250 ↳ (началом) и Target (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
254 ↳ искомой связи.</param>
255 /// <param name="target">Индекс связи, которая является концом на искомой
256 ↳ связи.</param>
257 /// <returns>Индекс искомой связи.</returns>
258 public ulong Search(ulong source, ulong target)
259 {
260     var root = Header->FirstAsSource;
261     while (root != 0)
262     {
263         var rootSource = Links[root].Source;
264         var rootTarget = Links[root].Target;
265         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
266             ↳ node.Key < root.Key
267         {
268             root = GetLeftOrDefault(root);
269         }
270         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
271             ↳ // node.Key > root.Key
272         {
273             root = GetRightOrDefault(root);
274         }
275         else // node.Key == root.Key
276         {
277             return root;
278         }
279     }
280     return 0;
281 }
282
283 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

277 private static bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
278     ↳ ulong secondSource, ulong secondTarget)
279     => firstSource < secondSource || (firstSource == secondSource && firstTarget
280     ↳ < secondTarget);
281
282 [MethodImpl(MethodImplOptions.AggressiveInlining)]
283 private static bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
284     ↳ ulong secondSource, ulong secondTarget)
285     => firstSource > secondSource || (firstSource == secondSource && firstTarget
286     ↳ > secondTarget);
287
288 [MethodImpl(MethodImplOptions.AggressiveInlining)]
289 protected override void ClearNode(ulong node)
290 {
291     Links[node].LeftAsSource = 0UL;
292     Links[node].RightAsSource = 0UL;
293     Links[node].SizeAsSource = 0UL;
294 }
295
296 [MethodImpl(MethodImplOptions.AggressiveInlining)]
297 protected override ulong GetZero() => 0UL;
298
299 [MethodImpl(MethodImplOptions.AggressiveInlining)]
300 protected override ulong GetOne() => 1UL;
301
302 [MethodImpl(MethodImplOptions.AggressiveInlining)]
303 protected override ulong GetTwo() => 2UL;
304
305 [MethodImpl(MethodImplOptions.AggressiveInlining)]
306 protected override bool ValueEqualToZero(IntPtr pointer) =>
307     ↳ *(ulong*)pointer.ToPointer() == 0UL;
308
309 [MethodImpl(MethodImplOptions.AggressiveInlining)]
310 protected override bool EqualToZero(ulong value) => value == 0UL;
311
312 [MethodImpl(MethodImplOptions.AggressiveInlining)]
313 protected override bool IsEquals(ulong first, ulong second) => first == second;
314
315 [MethodImpl(MethodImplOptions.AggressiveInlining)]
316 protected override bool GreaterThanZero(ulong value) => value > 0UL;
317
318 [MethodImpl(MethodImplOptions.AggressiveInlining)]
319 protected override bool GreaterThan(ulong first, ulong second) => first > second;
320
321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
322 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first
323     ↳ >= second;
324
325 [MethodImpl(MethodImplOptions.AggressiveInlining)]
326 protected override bool GreaterOrEqualThanZero(ulong value) => true; // value
327     ↳ >= 0 is always true for ulong
328
329 [MethodImpl(MethodImplOptions.AggressiveInlining)]
330 protected override bool LessOrEqualThanZero(ulong value) => value == 0; //
331     ↳ value is always >= 0 for ulong
332
333 [MethodImpl(MethodImplOptions.AggressiveInlining)]
334 protected override bool LessOrEqualThan(ulong first, ulong second) => first <=
335     ↳ second;
336
337 [MethodImpl(MethodImplOptions.AggressiveInlining)]
338 protected override bool LessThanZero(ulong value) => false; // value < 0 is always
339     ↳ false for ulong

```

```

330     [MethodImpl(MethodImplOptions.AggressiveInlining)]
331     protected override bool LessThan(ulong first, ulong second) => first < second;
332
333     [MethodImpl(MethodImplOptions.AggressiveInlining)]
334     protected override ulong Increment(ulong value) => ++value;
335
336     [MethodImpl(MethodImplOptions.AggressiveInlining)]
337     protected override ulong Decrement(ulong value) => --value;
338
339     [MethodImpl(MethodImplOptions.AggressiveInlining)]
340     protected override ulong Add(ulong first, ulong second) => first + second;
341
342     [MethodImpl(MethodImplOptions.AggressiveInlining)]
343     protected override ulong Subtract(ulong first, ulong second) => first - second;
344 }
345
346 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
347 {
348     public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
349         : base(memory)
350     {
351     }
352 }
353
354 //protected override IntPtr GetLeft(ulong node) => new
355     ↳ IntPtr(&Links[node].LeftAsTarget);
356
357 //protected override IntPtr GetRight(ulong node) => new
358     ↳ IntPtr(&Links[node].RightAsTarget);
359
360 //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
361
362 //protected override void SetLeft(ulong node, ulong left) =>
363     ↳ Links[node].LeftAsTarget = left;
364
365 //protected override void SetRight(ulong node, ulong right) =>
366     ↳ Links[node].RightAsTarget = right;
367
368 //protected override void SetSize(ulong node, ulong size) =>
369     ↳ Links[node].SizeAsTarget = size;
370
371 protected override IntPtr GetLeftPointer(ulong node) => new
372     ↳ IntPtr(&Links[node].LeftAsTarget);
373
374 protected override IntPtr GetRightPointer(ulong node) => new
375     ↳ IntPtr(&Links[node].RightAsTarget);
376
377 protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsTarget;
378
379 protected override ulong GetRightValue(ulong node) => Links[node].RightAsTarget;
380
381 protected override ulong GetSize(ulong node)
382 {
383     var previousValue = Links[node].SizeAsTarget;
384     //return MathHelpers.PartialRead(previousValue, 5, -5);
385     return (previousValue & 4294967264) >> 5;
386 }
387
388 protected override void SetLeft(ulong node, ulong left) =>
389     ↳ Links[node].LeftAsTarget = left;
390
391 protected override void SetRight(ulong node, ulong right) =>
392     ↳ Links[node].RightAsTarget = right;

```

```

384
385 protected override void SetSize(ulong node, ulong size)
386 {
387     var previousValue = Links[node].SizeAsTarget;
388     //var modified = MathHelpers.PartialWrite(previousValue, size, 5, -5);
389     var modified = (previousValue & 31) | ((size & 134217727) << 5);
390     Links[node].SizeAsTarget = modified;
391 }
392
393 protected override bool GetLeftIsChild(ulong node)
394 {
395     var previousValue = Links[node].SizeAsTarget;
396     //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
397     return (previousValue & 16) >> 4 == 1UL;
398     // TODO: Check if this is possible to use
399     //var nodeSize = GetSize(node);
400     //var left = GetLeftValue(node);
401     //var leftSize = GetSizeOrZero(left);
402     //return leftSize > 0 && nodeSize > leftSize;
403 }
404
405 protected override void SetLeftIsChild(ulong node, bool value)
406 {
407     var previousValue = Links[node].SizeAsTarget;
408     //var modified = MathHelpers.PartialWrite(previousValue,
409     ↳ (ulong)(Integer)value, 4, 1);
410     var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
411     Links[node].SizeAsTarget = modified;
412 }
413
414 protected override bool GetRightIsChild(ulong node)
415 {
416     var previousValue = Links[node].SizeAsTarget;
417     //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
418     return (previousValue & 8) >> 3 == 1UL;
419     // TODO: Check if this is possible to use
420     //var nodeSize = GetSize(node);
421     //var right = GetRightValue(node);
422     //var rightSize = GetSizeOrZero(right);
423     //return rightSize > 0 && nodeSize > rightSize;
424 }
425
426 protected override void SetRightIsChild(ulong node, bool value)
427 {
428     var previousValue = Links[node].SizeAsTarget;
429     //var modified = MathHelpers.PartialWrite(previousValue,
430     ↳ (ulong)(Integer)value, 3, 1);
431     var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
432     Links[node].SizeAsTarget = modified;
433 }
434
435 protected override sbyte GetBalance(ulong node)
436 {
437     var previousValue = Links[node].SizeAsTarget;
438     //var value = MathHelpers.PartialRead(previousValue, 0, 3);
439     var value = previousValue & 7;
440     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
441     ↳ 124 : value & 3);
442     return unpackedValue;
443 }
444
445 protected override void SetBalance(ulong node, sbyte value)

```

```

443 {
444     var previousValue = Links[node].SizeAsTarget;
445     var packagedValue = (ulong)(((byte)value >> 5) & 4) | value & 3;
446     //var modified = MathHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
447     var modified = (previousValue & 4294967288) | (packagedValue & 7);
448     Links[node].SizeAsTarget = modified;
449 }
450
451 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
452     => Links[first].Target < Links[second].Target ||
453     (Links[first].Target == Links[second].Target && Links[first].Source <
454         ↳ Links[second].Source);
455
456 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
457     => Links[first].Target > Links[second].Target ||
458     (Links[first].Target == Links[second].Target && Links[first].Source >
459         ↳ Links[second].Source);
460
461 protected override ulong GetTreeRoot() => Header->FirstAsTarget;
462
463 protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
464
465 [MethodImpl(MethodImplOptions.AggressiveInlining)]
466 protected override void ClearNode(ulong node)
467 {
468     Links[node].LeftAsTarget = 0UL;
469     Links[node].RightAsTarget = 0UL;
470     Links[node].SizeAsTarget = 0UL;
471 }
472 }

```

## ./Sequences/Converters/BalancedVariantConverter.cs

```

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

```

```

28     }
29     // Keep creating layer after layer
30     while (length > 2)
31     {
32         HalveSequence(sequence, sequence, length);
33         length = (length / 2) + (length % 2);
34     }
35     return Links.GetOrCreate(sequence[0], sequence[1]);
36 }
37
38 private void HalveSequence(IList<TLink> destination, IList<TLink> source, int
39     ↳ length)
40 {
41     var loopedLength = length - (length % 2);
42     for (var i = 0; i < loopedLength; i += 2)
43     {
44         destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
45     }
46     if (length > loopedLength)
47     {
48         destination[length / 2] = source[length - 1];
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
15     ↳ изолированно от Links на этапе сжатия.
16     /// А именно будет создаваться временный список пар необходимых для
17     ↳ выполнения сжатия, в таком случае тип значения элемента массива может быть
18     ↳ любым, как char так и ulong.
19     /// Как только список/словарь пар был выявлен можно разом выполнить
20     ↳ создание всех этих пар, а так же разом выполнить замену.
21     /// </remarks>
22     public class CompressingConverter<TLink> :
23         ↳ LinksListToSequenceConverterBase<TLink>
24     {
25         private static readonly LinksCombinedConstants<bool, TLink, long> _constants =
26             ↳ Default<LinksCombinedConstants<bool, TLink, long>>.Instance;
27         private static readonly EqualityComparer<TLink> _equalityComparer =
28             ↳ EqualityComparer<TLink>.Default;
29         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
30
31         private readonly IConverter<IList<TLink>, TLink> _baseConverter;
32         private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
33         private readonly TLink _minFrequencyToCompress;
34         private readonly bool _doInitialFrequenciesIncrement;
35         private Doublet<TLink> _maxDoublet;

```



```

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

```

```

83         return sequence;
84     }
85     if (sequence.Count == 2)
86     {
87         return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
88     }
89     // TODO: arraypool with min size (to improve cache locality) or stackalloc with
90     ↪ 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,
143                 ↪ maxDoubletTarget);

```

```

141     }
142     var maxDoubletReplacementLink = _maxDoubletData.Link;
143     oldLength--;
144     var oldLengthMinusTwo = oldLength - 1;
145     // Substitute all usages
146     int w = 0, r = 0; // (r == read, w == write)
147     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(
165                     ↪ y(maxDoubletReplacementLink,
166                     ↪ next);
167             }
168             copy[w++].Element = maxDoubletReplacementLink;
169             r++;
170             newLength--;
171         }
172         else
173         {
174             copy[w++] = copy[r];
175         }
176     }
177     if (w < newLength)
178     {
179         copy[w] = copy[r];
180     }
181     oldLength = newLength;
182     ResetMaxDoublet();
183     UpdateMaxDoublet(copy, newLength);
184 }
185 return newLength;
186 }
187
188 [MethodImpl(MethodImplOptions.AggressiveInlining)]
189 private void ResetMaxDoublet()
190 {
191     _maxDoublet = new Doublet<TLink>();
192     _maxDoubletData = new LinkFrequency<TLink>();
193 }
194
195 [MethodImpl(MethodImplOptions.AggressiveInlining)]
196 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
197 {
198     Doublet<TLink> doublet = default;
199     for (var i = 1; i < length; i++)
200     {
201         doublet.Source = copy[i - 1].Element;
202         doublet.Target = copy[i].Element;
203     }

```

```

198         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
199     }
200 }
201
202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 private void UpdateMaxDoublet(ref Doublet<TLink> doublet,
204     ↪ LinkFrequency<TLink> data)
205 {
206     var frequency = data.Frequency;
207     var maxFrequency = _maxDoubletData.Frequency;
208     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
209     ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives
210     ↪ better compression string data (and gives collisions quickly) */
211     ↪ maxDoublet.Source + _maxDoublet.Target)))
212     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
213     ↪ (_comparer.Compare(maxFrequency, frequency) < 0 ||
214     ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
215     ↪ _comparer.Compare(ArithmeticHelpers.Add(doublet.Source,
216     ↪ doublet.Target), ArithmeticHelpers.Add(_maxDoublet.Source,
217     ↪ _maxDoublet.Target)) > 0))) /* gives better stability and better
218     ↪ compression on sequent data and even on random numbers data (but gives
219     ↪ collisions anyway) */
220     {
221         _maxDoublet = doublet;
222         _maxDoubletData = data;
223     }
224 }
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> :
7         ↪ IConverter<IList<TLink>, TLink>
8     {
9         protected readonly IList<TLink> Links;
10         public LinksListToSequenceConverterBase(IList<TLink> links) => Links = links;
11         public abstract TLink Convert(IList<TLink> source);
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> :
8         ↪ LinksListToSequenceConverterBase<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         private readonly IConverter<IList<TLink>>
15             ↪ _sequenceToItsLocalElementLevelsConverter;

```

```

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

```

```

72     }
73     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
94     ↳ previous, TLink current, TLink next)
95 {
96     return _comparer.Compare(previous, next) > 0
97         ? _comparer.Compare(previous, current) < 0 ? previous : current
98         : _comparer.Compare(next, current) < 0 ? next : current;
99 }
100
101 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink
102     ↳ next) => _comparer.Compare(next, current) < 0 ? next : current;
103
104 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink
105     ↳ current) => _comparer.Compare(previous, current) < 0 ? previous : current;
106 }

```

## ./Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs

```

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

```

#### ./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs

```

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

```

#### ./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;
11         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
12         ↪ IList<TLink>>>> duplicateFragmentsProvider) =>
13         ↪ _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> :
17         ↳ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
18         ↳ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
19     {
20         private readonly ILinks<TLink> _links;
21         private readonly ISequences<TLink> _sequences;
22         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
23         private BitString _visited;
24
25         private class ItemEqualityComparer :
26             ↳ IEqualityComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
27         {
28             private readonly IListEqualityComparer<TLink> _listComparer;
29             public ItemEqualityComparer() => _listComparer =
30                 ↳ Default<IListEqualityComparer<TLink>>.Instance;
31             public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
32                 ↳ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
33                 ↳ _listComparer.Equals(left.Key, right.Key) &&
34                 ↳ _listComparer.Equals(left.Value, right.Value);
35             public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
36                 ↳ HashHelpers.Generate(_listComparer.GetHashCode(pair.Key),
37                 ↳ _listComparer.GetHashCode(pair.Value));
38         }
39
40         private class ItemComparer : IComparer<KeyValuePair<IList<TLink>,
41             ↳ IList<TLink>>>
42         {
43             private readonly IListComparer<TLink> _listComparer;
44
45             public ItemComparer() => _listComparer =
46                 ↳ Default<IListComparer<TLink>>.Instance;
47
48             public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
49                 ↳ KeyValuePair<IList<TLink>, IList<TLink>> right)
50             {
51                 var intermediateResult = _listComparer.Compare(left.Key, right.Key);
52                 if (intermediateResult == 0)
53                 {
54                     intermediateResult = _listComparer.Compare(left.Value, right.Value);
55                 }
56                 return intermediateResult;
57             }
58         }
59     }
60 }

```

```

47
48 public DuplicateSegmentsProvider(ILinks<TLink> links, ISequences<TLink>
49     ↳ sequences)
50     : base(minimumStringSegmentLength: 2)
51 {
52     _links = links;
53     _sequences = sequences;
54 }
55
56 public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
57 {
58     _groups = new HashSet<KeyValuePair<IList<TLink>,
59         ↳ IList<TLink>>>(Default<ItemEqualityComparer>.Instance);
60     var count = _links.Count();
61     _visited = new BitString((long)(Integer<TLink>)count + 1);
62     _links.Each(link =>
63     {
64         var linkIndex = _links.GetIndex(link);
65         var linkBitIndex = (long)(Integer<TLink>)linkIndex;
66         if (!_visited.Get(linkBitIndex))
67         {
68             var sequenceElements = new List<TLink>();
69             _sequences.EachPart(sequenceElements.AddAndReturnTrue, linkIndex);
70             if (sequenceElements.Count > 2)
71             {
72                 WalkAll(sequenceElements);
73             }
74             return _links.Constants.Continue;
75         });
76     var resultList = _groups.ToList();
77     var comparer = Default<ItemComparer>.Instance;
78     resultList.Sort(comparer);
79     #if DEBUG
80     foreach (var item in resultList)
81     {
82         PrintDuplicates(item);
83     }
84     #endif
85     return resultList;
86 }
87
88 protected override Segment<TLink> CreateSegment(IList<TLink> elements, int
89     ↳ offset, int length) => new Segment<TLink>(elements, offset, length);
90
91 protected override void OnDuplicateFound(Segment<TLink> segment)
92 {
93     var duplicates = CollectDuplicatesForSegment(segment);
94     if (duplicates.Count > 1)
95     {
96         _groups.Add(new KeyValuePair<IList<TLink>,
97             ↳ IList<TLink>>(segment.ToArray(), duplicates));
98     }
99 }
100
101 private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
102 {
103     var duplicates = new List<TLink>();
104     var readAsElement = new HashSet<TLink>();
105     _sequences.Each(sequence =>
106     {
107         duplicates.Add(sequence);
108     });
109 }

```

```

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.GetAllPartiallyMatchingSequences(
120             es4((HashSet<ulong>)(object)readAsElement,
121                 => (IList<ulong>)segment);
122             foreach (var partiallyMatchedSequence in partiallyMatched)
123             {
124                 TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
125                 duplicates.Add(sequenceIndex);
126             }
127         }
128     }
129     duplicates.Sort();
130     return duplicates;
131 }
132
133 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>>
134     => duplicatesItem)
135 {
136     if (!(_links is ILinks<ulong> ulongLinks))
137     {
138         return;
139     }
140     var duplicatesKey = duplicatesItem.Key;
141     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
142     Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)}");
143     var duplicatesList = duplicatesItem.Value;
144     for (int i = 0; i < duplicatesList.Count; i++)
145     {
146         ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
147         var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x
148             => Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
149             => UnicodeMap.IsCharLink(link.Index) ?
150             => sb.Append(UnicodeMap.FromLinkToChar(link.Index)) :
151             => sb.Append(link.Index));
152         Console.WriteLine(formattedSequenceStructure);
153         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
154             => ulongLinks);
155         Console.WriteLine(sequenceString);
156     }
157     Console.WriteLine();
158 }
159 }
160 }
161 }

```

#### ./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache

```

```

5 {
6     public class FrequenciesCacheBasedLinkFrequencyIncrementer<TLink> :
7         => Incrementer<IList<TLink>>
8     {
9         private readonly LinkFrequenciesCache<TLink> _cache;
10
11         public FrequenciesCacheBasedLinkFrequencyIncrementer(LinkFrequenciesCache<TLink>
12             => ink> cache) => _cache =
13             => cache;
14
15         /// <remarks>Sequence itseft is not changed, only frequency of its doublets is
16         => incremented.</remarks>
17         public IList<TLink> Increment(IList<TLink> sequence)
18         {
19             _cache.IncrementFrequencies(sequence);
20             return sequence;
21         }
22     }
23 }

```

#### ./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<TLink> cache) => _cache =
11             => cache;
12
13         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref
14             => source).Frequency;
15     }
16 }

```

#### ./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
11     /// data 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>>
21             => doubletsCache;
22         private readonly ICounter<TLink, TLink> _frequencyCounter;
23     }
24 }

```

```

21 public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink>
    ↪ frequencyCounter)
22 : base(links)
23 {
24     _doubletsCache = new Dictionary<Doublet<TLink>,
    ↪ LinkFrequency<TLink>>(4096, DoubletComparer<TLink>.Default);
25     _frequencyCounter = frequencyCounter;
26 }
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
30 {
31     var doublet = new Doublet<TLink>(source, target);
32     return GetFrequency(ref doublet);
33 }
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
37 {
38     _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
39     return data;
40 }
41
42 public void IncrementFrequencies(ICollection<TLink> sequence)
43 {
44     for (var i = 1; i < sequence.Count; i++)
45     {
46         IncrementFrequency(sequence[i - 1], sequence[i]);
47     }
48 }
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
52 {
53     var doublet = new Doublet<TLink>(source, target);
54     return IncrementFrequency(ref doublet);
55 }
56
57 public void PrintFrequencies(ICollection<TLink> sequence)
58 {
59     for (var i = 1; i < sequence.Count; i++)
60     {
61         PrintFrequency(sequence[i - 1], sequence[i]);
62     }
63 }
64
65 public void PrintFrequency(TLink source, TLink target)
66 {
67     var number = GetFrequency(source, target).Frequency;
68     Console.WriteLine("{0},{1} - {2}", source, target, number);
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
73 {
74     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
75     {
76         data.IncrementFrequency();
77     }
78     else
79     {
80         var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
81         data = new LinkFrequency<TLink>(Integer<TLink>.One, link);

```

```

82         if (!_equalityComparer.Equals(link, default))
83         {
84             data.Frequency = ArithmeticHelpers.Add(data.Frequency,
    ↪ _frequencyCounter.Count(link));
85         }
86         _doubletsCache.Add(doublet, data);
87     }
88     return data;
89 }
90
91 public void ValidateFrequencies()
92 {
93     foreach (var entry in _doubletsCache)
94     {
95         var value = entry.Value;
96         var linkIndex = value.Link;
97         if (!_equalityComparer.Equals(linkIndex, default))
98         {
99             var frequency = value.Frequency;
100             var count = _frequencyCounter.Count(linkIndex);
101             // TODO: Why `frequency` always greater than `count` by 1?
102             if (((_comparer.Compare(frequency, count) > 0) &&
    ↪ (_comparer.Compare(ArithmeticHelpers.Subtract(frequency, count),
    ↪ Integer<TLink>.One) > 0))
103                 || ((_comparer.Compare(count, frequency) > 0) &&
    ↪ (_comparer.Compare(ArithmeticHelpers.Subtract(count, frequency),
    ↪ Integer<TLink>.One) > 0)))
104             {
105                 throw new InvalidOperationException("Frequencies validation failed.");
106             }
107         }
108         //else
109         // {
110         //     if (value.Frequency > 0)
111         //     {
112         //         var frequency = value.Frequency;
113         //         linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
114         //         var count = _countLinkFrequency(linkIndex);
115         //         if ((frequency > count && frequency - count > 1) || (count > frequency
116         //     ↪ && count - frequency > 1))
117         //         throw new Exception("Frequencies validation failed.");
118         //     }
119         // }
120     }
121 }
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 =
        ↳ ArithmeticHelpers<TLink>.Increment(Frequency);
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     public void DecrementFrequency() => Frequency =
        ↳ ArithmeticHelpers<TLink>.Decrement(Frequency);
24
25     public override string ToString() => $"F: {Frequency}, L: {Link}";
26 }
27 }

```

#### ./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs

```

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

```

#### ./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 =
        ↳ EqualityComparer<TLink>.Default;
11         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13         protected readonly ILinks<TLink> _links;
14         protected readonly TLink _sequenceLink;
15         protected readonly TLink _symbol;

```

```

16         protected TLink _total;
17
18         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink
        ↳ sequenceLink, TLink symbol)
19         {
20             _links = links;
21             _sequenceLink = sequenceLink;
22             _symbol = symbol;
23             _total = default;
24         }
25
26         public virtual TLink Count()
27         {
28             if (_comparer.Compare(_total, default) > 0)
29             {
30                 return _total;
31             }
32             StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource,
        ↳ _links.GetTarget, IsElement, VisitElement);
        return _total;
33     }
34
35     private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
        ↳ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher
        ↳ instead of IsPartialPoint
36
37     private bool VisitElement(TLink element)
38     {
39         if (_equalityComparer.Equals(element, _symbol))
40         {
41             _total = ArithmeticHelpers.Increment(_total);
42         }
43         return true;
44     }
45 }
46 }
47 }

```

#### ./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,
        ↳ ICriteriaMatcher<TLink> markedSequenceMatcher)
11         {
12             _links = links;
13             _markedSequenceMatcher = markedSequenceMatcher;
14         }
15
16         public TLink Count(TLink argument) => new
        ↳ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
        ↳ _markedSequenceMatcher, argument).Count();
17     }
18 }

```



## ./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,
13             ↳ symbol)
14             => _markedSequenceMatcher = markedSequenceMatcher;
15
16         protected override void CountSequenceSymbolFrequency(TLink link)
17         {
18             var symbolFrequencyCounter = new
19                 ↳ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
20                 ↳ _markedSequenceMatcher, link, _symbol);
21             _total = ArithmeticHelpers.Add(_total, symbolFrequencyCounter.Count());
22         }
23     }
24 }
```

## ./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 =
9             ↳ links;
10         public TLink Count(TLink symbol) => new
11             ↳ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
12             ↳ symbol).Count();
13     }
14 }
```

## ./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
19             ↳ symbol)
20         {
```

```
21             _links = links;
22             _symbol = symbol;
23             _visits = new HashSet<TLink>();
24             _total = default;
25         }
26
27         public TLink Count()
28         {
29             if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
30             {
31                 return _total;
32             }
33             CountCore(_symbol);
34             return _total;
35         }
36
37         private void CountCore(TLink link)
38         {
39             var any = _links.Constants.Any;
40             if (_equalityComparer.Equals(_links.Count(any, link), default))
41             {
42                 CountSequenceSymbolFrequency(link);
43             }
44             else
45             {
46                 _links.Each(EachElementHandler, any, link);
47             }
48         }
49
50         protected virtual void CountSequenceSymbolFrequency(TLink link)
51         {
52             var symbolFrequencyCounter = new
53                 ↳ SequenceSymbolFrequencyOneOffCounter<TLink>(_links, link, _symbol);
54             _total = ArithmeticHelpers.Add(_total, symbolFrequencyCounter.Count());
55         }
56
57         private TLink EachElementHandler(IList<TLink> doublet)
58         {
59             {
60                 var constants = _links.Constants;
61                 var doubletIndex = doublet[constants.IndexPart];
62                 if (_visits.Add(doubletIndex))
63                 {
64                     CountCore(doubletIndex);
65                 }
66             }
67             return constants.Continue;
68         }
69     }
70 }
```

## ./Sequences/HeightProviders/CachedSequenceHeightProvider.cs

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

```

11 private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
12 private readonly IConverter<TLink> _addressToUnaryNumberConverter;
13 private readonly IConverter<TLink> _unaryNumberToAddressConverter;
14 private readonly IPropertyOperator<TLink, TLink, TLink> _propertyOperator;
15
16 public CachedSequenceHeightProvider(
17     ILinks<TLink> links,
18     ISequenceHeightProvider<TLink> baseHeightProvider,
19     IConverter<TLink> addressToUnaryNumberConverter,
20     IConverter<TLink> unaryNumberToAddressConverter,
21     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> :
7         ↳ LinksOperatorBase<TLink>, ISequenceHeightProvider<TLink>
8     {
9         private readonly ICriteriaMatcher<TLink> _elementMatcher;
10
11         public DefaultSequenceRightHeightProvider(ILinks<TLink> links,
12             ↳ ICriteriaMatcher<TLink> elementMatcher) : base(links) => _elementMatcher
13             ↳ = elementMatcher;
14
15         public TLink Get(TLink sequence)
16         {
17             var height = default(TLink);
18             var pairOrElement = sequence;
19             while (!_elementMatcher.IsMatched(pairOrElement))
20             {
21                 pairOrElement = Links.GetTarget(pairOrElement);
22                 height = ArithmeticHelpers.Increment(height);
23             }
24         }
25     }
26 }

```

```

20     }
21     return height;
22 }
23 }
24 }

```

#### ./Sequences/HeightProviders/ISequenceHeightProvider.cs

```

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

```

#### ./Sequences/Sequences.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Runtime.CompilerServices;
5 using Platform.Collections;
6 using Platform.Collections.Lists;
7 using Platform.Threading.Synchronization;
8 using Platform.Helpers.Singletons;
9 using LinkIndex = System.UInt64;
10 using Platform.Data.Constants;
11 using Platform.Data.Sequences;
12 using Platform.Data.Doublets.Sequences.Walkers;
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     /// <summary>
17     /// Представляет коллекцию последовательностей связей.
18     </summary>
19     <remarks>
20     /// Обязательно реализовать атомарность каждого публичного метода.
21     ///
22     /// TODO:
23     ///
24     /// !!! Повышение вероятности повторного использования групп
25     ↳ (подпоследовательностей),
26     /// через естественную группировку по unicode типам, все whitespace вместе, все
27     ↳ символы вместе, все числа вместе и т.п.
28     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность
29     ↳ (глубину графа)
30     ///
31     /// x*y - найти все связи между, в последовательностях любой формы, если не стоит
32     ↳ ограничитель на то, что является последовательностью, а что нет,
33     /// то находятся любые структуры связей, которые содержат эти элементы именно в
34     ↳ таком порядке.
35     ///
36     /// Рост последовательности слева и справа.
37     /// Поиск со звёздочкой.
38     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
39     /// так же проблема может быть решена при реализации дистанционных триггеров.
40     /// Нужны ли уникальные указатели вообще?
41     /// Что если обращение к информации будет происходить через содержимое всегда?
42     ///
43     /// Писать тесты.
44     ///
45 }

```

```

41  /// Можно убрать зависимость от конкретной реализации Links,
42  /// на зависимость от абстрактного элемента, который может быть представлен
43  ↪ несколькими способами.
44  ///
45  /// Можно ли как-то сделать один общий интерфейс
46  ///
47  /// Блокчейн и/или гит для распределённой записи транзакций.
48  /// </remarks>
49  /// </summary>
50  public partial class Sequences : ISequences<ulong> // IList<string>, IList<ulong[]>
51  ↪ (после завершения реализации Sequences)
52  {
53      private static readonly LinksCombinedConstants<bool, ulong, long> _constants =
54      ↪ Default<LinksCombinedConstants<bool, ulong, long>>.Instance;
55
56      /// <summary>Возвращает значение ulong, обозначающее любое количество
57      ↪ связей.</summary>
58      public const ulong ZeroOrMany = ulong.MaxValue;
59
60      public SequencesOptions<ulong> Options;
61      public readonly SynchronizedLinks<ulong> Links;
62      public readonly ISynchronization Sync;
63
64      public Sequences(SynchronizedLinks<ulong> links)
65      : this(links, new SequencesOptions<ulong>())
66      {
67      }
68
69      public Sequences(SynchronizedLinks<ulong> links, SequencesOptions<ulong> options)
70      {
71          Links = links;
72          Sync = links.SyncRoot;
73          Options = options;
74
75          Options.ValidateOptions();
76          Options.InitOptions(Links);
77      }
78
79      public bool IsSequence(ulong sequence)
80      {
81          return Sync.ExecuteReadOperation(() =>
82          {
83              if (Options.UseSequenceMarker)
84              {
85                  return Options.MarkedSequenceMatcher.IsMatched(sequence);
86              }
87              return !Links.Unsync.IsPartialPoint(sequence);
88          });
89      }
90
91      [MethodImpl(MethodImplOptions.AggressiveInlining)]
92      private ulong GetSequenceByElements(ulong sequence)
93      {
94          if (Options.UseSequenceMarker)
95          {
96              return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
97          }
98          return sequence;
99      }
100
101      private ulong GetSequenceElements(ulong sequence)
102      {

```

```

100      if (Options.UseSequenceMarker)
101      {
102          var linkContents = new UInt64Link(Links.GetLink(sequence));
103          if (linkContents.Source == Options.SequenceMarkerLink)
104          {
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,
122          ↪ _constants.Any);
123      }
124      if (sequence.Length == 1) // Первая связь это адрес
125      {
126          if (sequence[0] == _constants.Null)
127          {
128              return 0;
129          }
130          if (sequence[0] == _constants.Any)
131          {
132              return Count();
133          }
134          if (Options.UseSequenceMarker)
135          {
136              return Links.Count(_constants.Any, Options.SequenceMarkerLink,
137              ↪ sequence[0]);
138          }
139          return Links.Exists(sequence[0]) ? 1UL : 0;
140      }
141      throw new NotImplementedException();
142  }
143
144  private ulong CountReferences(params ulong[] restrictions)
145  {
146      if (restrictions.Length == 0)
147      {
148          return 0;
149      }
150      if (restrictions.Length == 1) // Первая связь это адрес
151      {
152          if (restrictions[0] == _constants.Null)
153          {
154              return 0;
155          }
156          if (Options.UseSequenceMarker)
157          {
158              var elementsLink = GetSequenceElements(restrictions[0]);
159              var sequenceLink = GetSequenceByElements(elementsLink);
160              if (sequenceLink != _constants.Null)
161              {

```

```

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
214     ↪ элементы)
215 }
216
217 #endregion
218
219 #region Each
220
221 public List<ulong> Each(params ulong[] sequence)
222 {

```

```

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
261     ↪ sequence 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     {
279         if (!StepLeft(innerHandler, sequence[sequence.Count - 2],
280         ↪ sequence[sequence.Count - 1]))
281         {
282             return false;
283         }
284     }
285 }

```

```

280     }
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 =>
303     ↳ TryStepRightUp(handler, right, rightStep));
304
305 private bool TryStepRightUp(Func<ulong, bool> handler, ulong right, ulong
306     ↳ stepFrom)
307 {
308     var upStep = stepFrom;
309     var firstSource = Links.Unsync.GetTarget(upStep);
310     while (firstSource != right && firstSource != upStep)
311     {
312         upStep = firstSource;
313         firstSource = Links.Unsync.GetSource(upStep);
314     }
315     if (firstSource == right)
316     {
317         return handler(stepFrom);
318     }
319     return true;
320 }
321
322 private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong right) =>
323     ↳ Links.Unsync.Each(_constants.Any, right, leftStep => TryStepLeftUp(handler,
324     ↳ left, leftStep));
325
326 private bool TryStepLeftUp(Func<ulong, bool> handler, ulong left, ulong stepFrom)
327 {
328     var upStep = stepFrom;
329     var firstTarget = Links.Unsync.GetSource(upStep);
330     while (firstTarget != left && firstTarget != upStep)
331     {
332         upStep = firstTarget;
333         firstTarget = Links.Unsync.GetTarget(upStep);
334     }
335     if (firstTarget == left)
336     {
337         return handler(stepFrom);
338     }
339     return true;
340 }
341
342 #endregion

```

```

338
339 #region Update
340
341 public ulong Update(ulong[] sequence, ulong[] newSequence)
342 {
343     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
344     {
345         return _constants.Null;
346     }
347     if (sequence.IsNullOrEmpty())
348     {
349         return Create(newSequence);
350     }
351     if (newSequence.IsNullOrEmpty())
352     {
353         Delete(sequence);
354         return _constants.Null;
355     }
356     return Sync.ExecuteWriteOperation(() =>
357     {
358         Links.EnsureEachLinkIsAnyOrExists(sequence);
359         Links.EnsureEachLinkExists(newSequence);
360         return UpdateCore(sequence, newSequence);
361     });
362 }
363
364 private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
365 {
366     ulong bestVariant;
367     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
368         ↳ !sequence.EqualTo(newSequence))
369     {
370         bestVariant = CompactCore(newSequence);
371     }
372     else
373     {
374         bestVariant = CreateCore(newSequence);
375     }
376     // TODO: Check all options only ones before loop execution
377     // Возможно нужно две версии Each, возвращающий фактические
378     ↳ последовательности и с маркером,
379     // или возможно даже возвращать и тот и тот вариант. С другой стороны все
380     ↳ варианты 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
397             ↳ UInt64Link(Links.GetLink(sequenceElements));
398         var sequenceLink = GetSequenceByElements(sequenceElements);
399         var newSequenceElements = GetSequenceElements(newSequence);

```

```

396     var newSequenceLink = GetSequenceByElements(newSequenceElements);
397     if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
398     {
399         if (sequenceLink != _constants.Null)
400         {
401             Links.Unsync.Merge(sequenceLink, newSequenceLink);
402         }
403         Links.Unsync.Merge(sequenceElements, newSequenceElements);
404     }
405     ClearGarbage(sequenceElementsContents.Source);
406     ClearGarbage(sequenceElementsContents.Target);
407 }
408 else
409 {
410     if (Options.UseSequenceMarker)
411     {
412         var sequenceElements = GetSequenceElements(sequence);
413         var sequenceLink = GetSequenceByElements(sequenceElements);
414         var newSequenceElements = GetSequenceElements(newSequence);
415         var newSequenceLink = GetSequenceByElements(newSequenceElements);
416         if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
417         {
418             if (sequenceLink != _constants.Null)
419             {
420                 Links.Unsync.Merge(sequenceLink, newSequenceLink);
421             }
422             Links.Unsync.Merge(sequenceElements, newSequenceElements);
423         }
424     }
425     else
426     {
427         if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
428         {
429             Links.Unsync.Merge(sequence, newSequence);
430         }
431     }
432 }
433 }
434 #endregion
435 #region Delete
436
437 public void Delete(params ulong[] sequence)
438 {
439     Sync.ExecuteWriteOperation(() =>
440     {
441         // TODO: Check all options only ones before loop execution
442         foreach (var linkToDelete in Each(sequence))
443         {
444             DeleteOneCore(linkToDelete);
445         }
446     });
447 }
448
449 private void DeleteOneCore(ulong link)
450 {
451     if (Options.UseGarbageCollection)
452     {
453         var sequenceElements = GetSequenceElements(link);
454         var sequenceElementsContents = new
455             ↳ 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 #endregion
494 #region Compactification
495
496 /// <remarks>
497 /// bestVariant можно выбирать по максимальному числу использований,
498 /// но балансированный позволяет гарантировать уникальность (если есть
499 ↳ возможность,
500 /// гарантировать его использование в других местах).
501
502 ///
503 /// Получается этот метод должен игнорировать
504 ↳ Options.EnforceSingleSequenceVersionOnWrite
505 /// </remarks>
506 public ulong Compact(params ulong[] sequence)
507 {
508     return Sync.ExecuteWriteOperation(() =>
509     {
510         if (sequence.IsNullOrEmpty())
511         {
512             return _constants.Null;
513         }
514         Links.EnsureEachLinkExists(sequence);
515         return CompactCore(sequence);
516     });

```

```

517 [MethodImpl(MethodImplOptions.AggressiveInlining)]
518 private ulong CompactCore(params ulong[] sequence) => UpdateCore(sequence,
519     ↳ sequence);
520
521 #endregion
522
523 #region Garbage Collection
524
525 /// <remarks>
526 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted
527   ↳ которое можно 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;
532
533 private void ClearGarbage(ulong link)
534 {
535     if (IsGarbage(link))
536     {
537         var contents = new UInt64Link(Links.GetLink(link));
538         Links.Unsync.Delete(link);
539         ClearGarbage(contents.Source);
540         ClearGarbage(contents.Target);
541     }
542 }
543
544 #endregion
545
546 #region Walkers
547
548 public bool EachPart(Func<ulong, bool> handler, ulong sequence)
549 {
550     return Sync.ExecuteReadOperation(() =>
551     {
552         var links = Links.Unsync;
553         var walker = new RightSequenceWalker<ulong>(links);
554         foreach (var part in walker.Walk(sequence))
555         {
556             if (!handler(links.GetIndex(part)))
557             {
558                 return false;
559             }
560         }
561         return true;
562     });
563 }
564
565 public class Matcher : RightSequenceWalker<ulong>
566 {
567     private readonly Sequences _sequences;
568     private readonly IList<LinkIndex> _patternSequence;
569     private readonly HashSet<LinkIndex> _linksInSequence;
570     private readonly HashSet<LinkIndex> _results;
571     private readonly Func<ulong, bool> _stopableHandler;
572     private readonly HashSet<ulong> _readAsElements;
573     private int _filterPosition;
574
575     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
576         ↳ HashSet<LinkIndex> results, Func<LinkIndex, bool> stopableHandler,
577         ↳ HashSet<LinkIndex> readAsElements = null)
578         : base(sequences.Links.Unsync)
579     {

```

```

576     _sequences = sequences;
577     _patternSequence = patternSequence;
578     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x
579     ↳ != _constants.Any && x != ZeroOrMany));
580     _results = results;
581     _stopableHandler = stopableHandler;
582     _readAsElements = readAsElements;
583 }
584
585 protected override bool IsElement(IList<ulong> link) => base.IsElement(link) ||
586     ( _readAsElements != null &&
587     ↳ _readAsElements.Contains(Links.GetIndex(link))) ||
588     ↳ _linksInSequence.Contains(Links.GetIndex(link));
589
590 public bool FullMatch(LinkIndex sequenceToMatch)
591 {
592     _filterPosition = 0;
593     foreach (var part in Walk(sequenceToMatch))
594     {
595         if (!FullMatchCore(Links.GetIndex(part)))
596         {
597             break;
598         }
599     }
600     return _filterPosition == _patternSequence.Count;
601 }
602
603 private bool FullMatchCore(LinkIndex element)
604 {
605     if ( _filterPosition == _patternSequence.Count)
606     {
607         _filterPosition = -2; // Длиннее чем нужно
608         return false;
609     }
610     if ( _patternSequence[_filterPosition] != _constants.Any
611     && element != _patternSequence[_filterPosition])
612     {
613         _filterPosition = -1;
614         return false; // Начинается/Продолжается иначе
615     }
616     _filterPosition++;
617     return true;
618 }
619
620 public void AddFullMatchedToResults(ulong sequenceToMatch)
621 {
622     if (FullMatch(sequenceToMatch))
623     {
624         _results.Add(sequenceToMatch);
625     }
626 }
627
628 public bool HandleFullMatched(ulong sequenceToMatch)
629 {
630     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
631     {
632         return _stopableHandler(sequenceToMatch);
633     }
634     return true;
635 }
636
637 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 ///

```

```

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>
        ↪ sequencesToMatch)
715 {
716     foreach (var sequenceToMatch in sequencesToMatch)
717     {
718         if (PartialMatch(sequenceToMatch))
719         {
720             _readAsElements.Add(sequenceToMatch);
721             _results.Add(sequenceToMatch);
722         }
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         ///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
47             ↳ быть меньше или равен stopAt");
48     }
49     #endif
50     if ((stopAt - startAt) == 0)
51     {
52         return new[] { sequence[startAt] };
53     }
54     if ((stopAt - startAt) == 1)
55     {
56         return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt],
57             ↳ sequence[stopAt]) };
58     }
59     var variants = new ulong[(ulong)MathHelpers.Catalan(stopAt - startAt)];
60     var last = 0;
61     for (var splitter = startAt; splitter < stopAt; splitter++)
62     {
63         var left = CreateAllVariants2Core(sequence, startAt, splitter);
64         var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
65         for (var i = 0; i < left.Length; i++)
66         {
67             for (var j = 0; j < right.Length; j++)
68             {
69                 var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
70                 if (variant == _constants.Null)
71                 {
72                     throw new NotImplementedException("Creation cancellation is not
73                         ↳ implemented.");
74                 }
75                 variants[last++] = variant;
76             }
77         }
78     }
79     return variants;
80 }
81
82 public List<ulong> CreateAllVariants1(params ulong[] sequence)
83 {
84     return Sync.ExecuteWriteOperation(() =>
85     {

```

```

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
105                 ↳ implemented.");
106         }
107         results.Add(link);
108         return results;
109     }
110     var innerSequenceLength = sequence.Length - 1;
111     var innerSequence = new ulong[innerSequenceLength];
112     for (var li = 0; li < innerSequenceLength; li++)
113     {
114         var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
115         if (link == _constants.Null)
116         {
117             throw new NotImplementedException("Creation cancellation is not
118                 ↳ implemented.");
119         }
120         for (var isi = 0; isi < li; isi++)
121         {
122             innerSequence[isi] = sequence[isi];
123         }
124         innerSequence[li] = link;
125         for (var isi = li + 1; isi < innerSequenceLength; isi++)
126         {
127             innerSequence[isi] = sequence[isi + 1];
128         }
129         CreateAllVariants1Core(innerSequence, results);
130     }
131     return results;
132 }
133
134 #endregion
135
136 public HashSet<ulong> Each1(params ulong[] sequence)
137 {
138     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
139     Each1(link =>
140     {
141         if (!visitedLinks.Contains(link))
142         {
143             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,
401         ↪ x =>
402         {
403             if (filterPosition == sequence.Length)
404             {
405                 filterPosition = -2; // Длиннее чем нужно
406                 return false;
407             }
408             if (x != sequence[filterPosition])
409             {
410                 filterPosition = -1;
411                 return false; // Начинается иначе
412             }
413             filterPosition++;
414             return true;
415         });
416         if (filterPosition == sequence.Length)
417         {
418             results.Add(result);
419         }
420     }
421     if (sequence.Length >= 2)
422     {
423         StepRight(handler, sequence[0], sequence[1]);
424     }
425     var last = sequence.Length - 2;
426     for (var i = 1; i < last; i++)
427     {
428         PartialStepRight(handler, sequence[i], sequence[i + 1]);
429     }
430     if (sequence.Length >= 3)
431     {
432         StepLeft(handler, sequence[sequence.Length - 2], sequence[sequence.Length
433         ↪ - 1]);
434     }
435     return results;
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],
470         ↪ sequence[i + 1]);
471     }
472     if (sequence.Length >= 3)
473     {
474         StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length -
475         ↪ 2], sequence[sequence.Length - 1]);
476     }
477     return results;
478 }
479
480 public const int MaxSequenceFormatSize = 200;
481
482 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[]
483     ↪ knownElements) => FormatSequence(sequenceLink, (sb, x) => sb.Append(x),
484     ↪ true, knownElements);
485
486 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
487     ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[]
488     ↪ knownElements) => Links.SyncRoot.ExecuteReadOperation(() =>
489     ↪ FormatSequence(Links.Unsync, sequenceLink, elementToString, insertComma,
490     ↪ knownElements));
491
492 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
493     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
494     ↪ LinkIndex[] knownElements)
495 {
496     var linksInSequence = new HashSet<ulong>(knownElements);
497     //var entered = new HashSet<ulong>();
498     var sb = new StringBuilder();
499     sb.Append('{');
500     if (links.Exists(sequenceLink))
501     {
502         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource,
503         ↪ links.GetTarget,
504         ↪ x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
505         ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
506         {
507             if (insertComma && sb.Length > 1)
508             {
509                 sb.Append(',');

```

```

500     }
501     // if (entered.Contains(element))
502     //{
503     //     sb.Append('{');
504     //     element.ToString(sb, element);
505     //     sb.Append('}');
506     //}
507     //else
508     element.ToString(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             element.ToString(sb, element);
544             sb.Append('}');
545         }
546         else
547         {
548             element.ToString(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,
    ↪ Links.Unsync.GetTarget,
    ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
    ↪ x =>
580             {
581                 if (filterPosition == (sequence.Length - 1))
582                 {
583                     return false;
584                 }
585                 if (filterPosition >= 0)
586                 {
587                     if (x == sequence[filterPosition + 1])
588                     {
589                         filterPosition++;
590                     }
591                     else
592                     {
593                         return false;
594                     }
595                 }
596                 if (filterPosition < 0)
597                 {
598                     if (x == sequence[0])
599                     {
600                         filterPosition = 0;
601                     }
602                 }
603                 return true;
604             }
605         });
606         if (filterPosition == (sequence.Length - 1))
607         {
608             filteredResults.Add(result);
609         }
610     }

```

```

611         return filteredResults;
612     }
613     return new List<ulong>();
614 });
615 }
616
617 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
618 {
619     return Sync.ExecuteReadOperation(() =>
620     {
621         if (sequence.Length > 0)
622         {
623             Links.EnsureEachLinkExists(sequence);
624             var results = new HashSet<ulong>();
625             for (var i = 0; i < sequence.Length; i++)
626             {
627                 AllUsagesCore(sequence[i], results);
628             }
629             var filteredResults = new HashSet<ulong>();
630             var matcher = new Matcher(this, sequence, filteredResults, null);
631             matcher.AddAllPartialMatchedToResults(results);
632             return filteredResults;
633         }
634         return new HashSet<ulong>();
635     });
636 }
637
638 public bool GetAllPartiallyMatchingSequences2(Func<ulong, bool> handler, params
↳ ulong[] sequence)
639 {
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, IList<ulong> sequence)
720 {
721     return Sync.ExecuteReadOperation(() =>
722     {
723         if (sequence.Count > 0)
724         {
725             Links.EnsureEachLinkExists(sequence);
726             var results = new HashSet<LinkIndex>();
727             //var nextResults = new HashSet<ulong>();
728             //for (var i = 0; i < sequence.Length; i++)
729             //{
730             //    AllUsagesCore(sequence[i], nextResults);
731             //    if (results.IsNullOrEmpty())
732             //    {

```

```

733 //      results = nextResults;
734 //      nextResults = new HashSet<ulong>();
735 //  }
736 //  else
737 //  {
738 //      results.IntersectWith(nextResults);
739 //      nextResults.Clear();
740 //  }
741 //}
742 var collector1 = new AllUsagesCollector1(Links.Unsync, results);
743 collector1.Collect(Links.Unsync.GetLink(sequence[0]));
744 var next = new HashSet<ulong>();
745 for (var i = 1; i < sequence.Count; i++)
746 {
747     var collector = new AllUsagesCollector1(Links.Unsync, next);
748     collector.Collect(Links.Unsync.GetLink(sequence[i]));
749
750     results.IntersectWith(next);
751     next.Clear();
752 }
753 var filteredResults = new HashSet<ulong>();
754 var matcher = new Matcher(this, sequence, filteredResults, null,
    ↪ readAsElements);
755 matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x
    ↪ => x)); // OrderBy is a Hack
756 return filteredResults;
757 }
758 return new HashSet<ulong>();
759 });
760 }
761 // Does not work
762 public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong>
    ↪ readAsElements, params ulong[] sequence)
763 {
764     var visited = new HashSet<ulong>();
765     var results = new HashSet<ulong>();
766     var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
    ↪ true; }, readAsElements);
767     var last = sequence.Length - 1;
768     for (var i = 0; i < last; i++)
769     {
770         PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
771     }
772     return results;
773 }
774 }
775 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
776 {
777     return Sync.ExecuteReadOperation(() =>
778     {
779         if (sequence.Length > 0)
780         {
781             Links.EnsureEachLinkExists(sequence);
782             //var firstElement = sequence[0];
783             //if (sequence.Length == 1)
784             //{
785                 //results.Add(firstElement);
786                 //return results;
787             //}
788             //if (sequence.Length == 2)
789             //{
790

```

```

791 //var doublet = links.SearchCore(firstElement, sequence[1]);
792 //if (doublet != Doublets.Links.Null)
793 //    results.Add(doublet);
794 return results;
795 }
796 var lastElement = sequence[sequence.Length - 1];
797 Func<ulong, bool> handler = x =>
798 {
799     if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
    ↪ results.Add(x);
800     return true;
801 };
802 if (sequence.Length >= 2)
803     StepRight(handler, sequence[0], sequence[1]);
804 var last = sequence.Length - 2;
805 for (var i = 1; i < last; i++)
806     PartialStepRight(handler, sequence[i], sequence[i + 1]);
807 if (sequence.Length >= 3)
808     StepLeft(handler, sequence[sequence.Length - 2],
    ↪ sequence[sequence.Length - 1]);
809 //if (sequence.Length == 1)
810 //{
811 //    throw new NotImplementedException(); // all sequences, containing
    ↪ this element?
812 //}
813 if (sequence.Length == 2)
814 {
815     var results = new List<ulong>();
816     PartialStepRight(results.Add, sequence[0], sequence[1]);
817     return results;
818 }
819 var matches = new List<List<ulong>>();
820 var last = sequence.Length - 1;
821 for (var i = 0; i < last; i++)
822 {
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],
    ↪ matches[1][k]);
836         if (merged.Count > 0)
837             matches = new List<List<ulong>> { merged };
838         else
839             return new List<ulong>();
840     }
841 }
842 if (matches.Count > 0)
843 {
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 return new List<ulong>();
870 });
871 }
872 /// <remarks>
873 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
874 /// </remarks>
875 public HashSet<ulong> AllUsages(ulong link)
876 {
877     return Sync.ExecuteReadOperation(() =>
878     {
879         var usages = new HashSet<ulong>();
880         AllUsagesCore(link, usages);
881         return usages;
882     });
883 }
884 // При сборе всех использований (последовательностей) можно сохранять
885     ↪ обратный путь к той связи с которой начинался поиск (STTTSSSTT),
886 // причём достаточно одного бита для хранения перехода влево или вправо
887 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
888 {
889     bool handler(ulong doublet)
890     {
891         if (usages.Add(doublet))
892         {
893             AllUsagesCore(doublet, usages);
894         }
895         return true;
896     }
897     Links.Unsync.Each(link, _constants.Any, handler);
898     Links.Unsync.Each(_constants.Any, link, handler);
899 }
900 public HashSet<ulong> AllBottomUsages(ulong link)
901 {
902     return Sync.ExecuteReadOperation(() =>
903     {
904         var visits = new HashSet<ulong>();
905         var usages = new HashSet<ulong>();

```

```

905         AllBottomUsagesCore(link, visits, usages);
906         return usages;
907     });
908 }
909 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits,
910     ↪ 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 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
931 {
932     if (Options.UseSequenceMarker)
933     {
934         var counter = new
935             ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
936             ↪ Options.MarkedSequenceMatcher, symbol);
937         return counter.Count();
938     }
939     else
940     {
941         var counter = new
942             ↪ TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links, symbol);
943         return counter.Count();
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     }
963     return true;
964 }

```



```

962         return Links.Unsync.Each(link, _constants.Any, handler)
963         && Links.Unsync.Each(_constants.Any, link, handler);
964     }
965
966     public void CalculateAllUsages(ulong[] totals)
967     {
968         var calculator = new AllUsagesCalculator(Links, totals);
969         calculator.Calculate();
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,
990         ↪ CalculateCore);
991
992         private bool CalculateCore(ulong link)
993         {
994             if (_totals[link] == 0)
995             {
996                 var total = 1UL;
997                 _totals[link] = total;
998                 var visitedChildren = new HashSet<ulong>();
999                 bool linkCalculator(ulong child)
1000                 {
1001                     if (link != child && visitedChildren.Add(child))
1002                     {
1003                         total += _totals[child] == 0 ? 1 : _totals[child];
1004                     }
1005                     return true;
1006                 }
1007                 _links.Unsync.Each(link, _constants.Any, linkCalculator);
1008                 _links.Unsync.Each(_constants.Any, link, linkCalculator);
1009                 _totals[link] = total;
1010             }
1011             return true;
1012         }
1013     }
1014
1015     private class AllUsagesCalculator2
1016     {
1017         private readonly SynchronizedLinks<ulong> _links;
1018         private readonly ulong[] _totals;
1019
1020         public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1021         {
1022             _links = links;
1023             _totals = totals;
1024         }

```

```

1025     public void Calculate() => _links.Each(_constants.Any, _constants.Any,
1026     ↪ CalculateCore);
1027
1028     private bool IsElement(ulong link)
1029     {
1030         // _linksInSequence.Contains(link) ||
1031         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link)
1032         ↪ == link;
1033     }
1034
1035     private bool CalculateCore(ulong link)
1036     {
1037         // TODO: Проработать защиту от заикливания
1038         // Основано на SequenceWalker.WalkLeft
1039         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1040         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1041         Func<ulong, bool> isElement = IsElement;
1042         void visitLeaf(ulong parent)
1043         {
1044             if (link != parent)
1045             {
1046                 _totals[parent]++;
1047             }
1048         }
1049         void visitNode(ulong parent)
1050         {
1051             if (link != parent)
1052             {
1053                 _totals[parent]++;
1054             }
1055         }
1056         var stack = new Stack();
1057         var element = link;
1058         if (isElement(element))
1059         {
1060             visitLeaf(element);
1061         }
1062         else
1063         {
1064             while (true)
1065             {
1066                 if (isElement(element))
1067                 {
1068                     if (stack.Count == 0)
1069                     {
1070                         break;
1071                     }
1072                     element = stack.Pop();
1073                     var source = getSource(element);
1074                     var target = getTarget(element);
1075                     // Обработка элемента
1076                     if (isElement(target))
1077                     {
1078                         visitLeaf(target);
1079                     }
1080                     if (isElement(source))
1081                     {
1082                         visitLeaf(source);
1083                     }
1084                     element = source;
1085                 }

```

```

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 private class AllUsagesIntersectingCollector
1166 {
1167     private readonly SynchronizedLinks<ulong> _links;
1168     private readonly HashSet<ulong> _intersectWith;
1169     private readonly HashSet<ulong> _usages;
1170     private readonly HashSet<ulong> _enter;
1171
1172     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links,
1173     ↪ HashSet<ulong> intersectWith, HashSet<ulong> usages)
1174     {
1175         _links = links;
1176         _intersectWith = intersectWith;
1177         _usages = usages;
1178         _enter = new HashSet<ulong>(); // защита от зацикливания
1179     }
1180
1181     public bool Collect(ulong link)
1182     {
1183         if (_enter.Add(link))
1184         {
1185             if (_intersectWith.Contains(link))
1186             {
1187                 _usages.Add(link);
1188                 _links.Unsync.Each(link, _constants.Any, Collect);
1189                 _links.Unsync.Each(_constants.Any, link, Collect);
1190             }
1191             return true;
1192         }
1193     }
1194
1195     private void CloseInnerConnections(Action<ulong> handler, ulong left, ulong right)
1196     {
1197         TryStepLeftUp(handler, left, right);
1198         TryStepRightUp(handler, right, left);
1199     }
1200
1201     private void AllCloseConnections(Action<ulong> handler, ulong left, ulong right)
1202     {
1203         // Direct
1204         if (left == right)
1205         {
1206             handler(left);
1207         }
1208         var doublet = Links.Unsync.SearchOrDefault(left, right);
1209         if (doublet != _constants.Null)

```

```

1210     {
1211         handler(doublet);
1212     }
1213     // Inner
1214     CloseInnerConnections(handler, left, right);
1215     // Outer
1216     StepLeft(handler, left, right);
1217     StepRight(handler, left, right);
1218     PartialStepRight(handler, left, right);
1219     PartialStepLeft(handler, left, right);
1220 }
1221
1222 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
1223     ↳ HashSet<ulong> previousMatchings, long startAt)
1224 {
1225     if (startAt >= sequence.Length) // ?
1226     {
1227         return previousMatchings;
1228     }
1229     var secondLinkUsages = new HashSet<ulong>();
1230     AllUsagesCore(sequence[startAt], secondLinkUsages);
1231     secondLinkUsages.Add(sequence[startAt]);
1232     var matchings = new HashSet<ulong>();
1233     //for (var i = 0; i < previousMatchings.Count; i++)
1234     foreach (var secondLinkUsage in secondLinkUsages)
1235     {
1236         foreach (var previousMatching in previousMatchings)
1237         {
1238             // AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1239             ↳ secondLinkUsage);
1240             StepRight(matchings.AddAndReturnVoid, previousMatching,
1241             ↳ secondLinkUsage);
1242             TryStepRightUp(matchings.AddAndReturnVoid, secondLinkUsage,
1243             ↳ previousMatching);
1244             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1245             ↳ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1246             ↳ желаемым результатам.
1247             PartialStepRight(matchings.AddAndReturnVoid, previousMatching,
1248             ↳ secondLinkUsage);
1249         }
1250     }
1251     if (matchings.Count == 0)
1252     {
1253         return matchings;
1254     }
1255     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1);
1256     ↳ // ??
1257 }
1258
1259 private static void
1260     ↳ EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong> links,
1261     ↳ params ulong[] sequence)
1262 {
1263     if (sequence == null)
1264     {
1265         return;
1266     }
1267     for (var i = 0; i < sequence.Length; i++)
1268     {
1269         if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany &&
1270             ↳ !links.Exists(sequence[i]))

```

```

1260     {
1261         throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1262             ↳ $"patternSequence[{i}]");
1263     }
1264 }
1265
1266 // Pattern Matching -> Key To Triggers
1267 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1268 {
1269     return Sync.ExecuteReadOperation(() =>
1270     {
1271         patternSequence = Simplify(patternSequence);
1272         if (patternSequence.Length > 0)
1273         {
1274             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1275             var uniqueSequenceElements = new HashSet<ulong>();
1276             for (var i = 0; i < patternSequence.Length; i++)
1277             {
1278                 if (patternSequence[i] != _constants.Any && patternSequence[i] !=
1279                     ↳ ZeroOrMany)
1280                 {
1281                     uniqueSequenceElements.Add(patternSequence[i]);
1282                 }
1283             }
1284             var results = new HashSet<ulong>();
1285             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1286             {
1287                 AllUsagesCore(uniqueSequenceElement, results);
1288             }
1289             var filteredResults = new HashSet<ulong>();
1290             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1291             matcher.AddAllPatternMatchedToResults(results);
1292             return filteredResults;
1293         }
1294         return new HashSet<ulong>();
1295     });
1296 }
1297
1298 // Найти все возможные связи между указанным списком связей.
1299 // Находит связи между всеми указанными связями в любом порядке.
1300 // TODO: решить что делать с повторами (когда одни и те же элементы
1301 ↳ встречаются несколько раз в последовательности)
1302 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1303 {
1304     return Sync.ExecuteReadOperation(() =>
1305     {
1306         var results = new HashSet<ulong>();
1307         if (linksToConnect.Length > 0)
1308         {
1309             Links.EnsureEachLinkExists(linksToConnect);
1310             AllUsagesCore(linksToConnect[0], results);
1311             for (var i = 1; i < linksToConnect.Length; i++)
1312             {
1313                 var next = new HashSet<ulong>();
1314                 AllUsagesCore(linksToConnect[i], next);
1315                 results.IntersectWith(next);
1316             }
1317         }
1318         return results;
1319     });

```

```

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                 return results;
1339             }
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                 return results;
1364             }
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++)

```

```

1379     {
1380
1381         {
1382             var next = new BitString((long)Links.Unsync.Count() + 1); //new
1383             ⇨ BitArray((int) _links.Total + 1);
1384             var collector = new AllUsagesCollector2(Links.Unsync, next);
1385             collector.Collect(linksToConnect[i]);
1386             results = results.And(next);
1387         }
1388     }
1389     return results.GetSetUInt64Indices();
1390 });
1391 }
1392
1393 private static ulong[] Simplify(ulong[] sequence)
1394 {
1395     // Считаем новый размер последовательности
1396     long newLength = 0;
1397     var zeroOrManyStepped = false;
1398     for (var i = 0; i < sequence.Length; i++)
1399     {
1400         if (sequence[i] == ZeroOrMany)
1401         {
1402             if (zeroOrManyStepped)
1403             {
1404                 continue;
1405             }
1406             zeroOrManyStepped = true;
1407         }
1408         else
1409         {
1410             //if (zeroOrManyStepped) Is it efficient?
1411             zeroOrManyStepped = false;
1412         }
1413         newLength++;
1414     }
1415     // Строим новую последовательность
1416     zeroOrManyStepped = false;
1417     var newSequence = new ulong[newLength];
1418     long j = 0;
1419     for (var i = 0; i < sequence.Length; i++)
1420     {
1421         //var current = zeroOrManyStepped;
1422         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1423         //if (current && zeroOrManyStepped)
1424         //    continue;
1425         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1426         //if (zeroOrManyStepped && newZeroOrManyStepped)
1427         //    continue;
1428         //zeroOrManyStepped = newZeroOrManyStepped;
1429         if (sequence[i] == ZeroOrMany)
1430         {
1431             if (zeroOrManyStepped)
1432             {
1433                 continue;
1434             }
1435             zeroOrManyStepped = true;
1436         }
1437         else
1438         {
1439             //if (zeroOrManyStepped) Is it efficient?
1440             zeroOrManyStepped = false;
1441         }
1442         newSequence[j++] = sequence[i];

```

```

1440     }
1441     return newSequence;
1442 }
1443
1444 public static void TestSimplify()
1445 {
1446     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
        ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
        var simplifiedSequence = Simplify(sequence);
1447 }
1448
1449 public List<ulong> GetSimilarSequences() => new List<ulong>();
1450
1451 public void Prediction()
1452 {
1453     // links
1454     // sequences
1455 }
1456
1457 #region From Triplets
1458
1459 //public static void DeleteSequence(Link sequence)
1460 //{
1461 //}
1462 //}
1463
1464 public List<ulong> CollectMatchingSequences(ulong[] links)
1465 {
1466     if (links.Length == 1)
1467     {
1468         throw new Exception("Подпоследовательности с одним элементом не
        ↪ поддерживаются.");
1469     }
1470     var leftBound = 0;
1471     var rightBound = links.Length - 1;
1472     var left = links[leftBound++];
1473     var right = links[rightBound--];
1474     var results = new List<ulong>();
1475     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1476     return results;
1477 }
1478
1479 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
        ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1480 {
1481     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1482     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1483     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1484     {
1485         var nextLeftLink = middleLinks[leftBound];
1486         var elements = GetRightElements(leftLink, nextLeftLink);
1487         if (leftBound <= rightBound)
1488         {
1489             for (var i = elements.Length - 1; i >= 0; i--)
1490             {
1491                 var element = elements[i];
1492                 if (element != 0)
1493                 {
1494                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
        ↪ rightLink, rightBound, ref results);
1495                 }
1496             }
1497         }
1498     }
        else

```

```

1499     {
1500         for (var i = elements.Length - 1; i >= 0; i--)
1501         {
1502             var element = elements[i];
1503             if (element != 0)
1504             {
1505                 results.Add(element);
1506             }
1507         }
1508     }
1509 }
1510 else
1511 {
1512     var nextRightLink = middleLinks[rightBound];
1513     var elements = GetLeftElements(rightLink, nextRightLink);
1514     if (leftBound <= rightBound)
1515     {
1516         for (var i = elements.Length - 1; i >= 0; i--)
1517         {
1518             var element = elements[i];
1519             if (element != 0)
1520             {
1521                 CollectMatchingSequences(leftLink, leftBound, middleLinks, elements[i],
        ↪ rightBound - 1, ref results);
1522             }
1523         }
1524     }
1525 }
1526 else
1527 {
1528     for (var i = elements.Length - 1; i >= 0; i--)
1529     {
1530         var element = elements[i];
1531         if (element != 0)
1532         {
1533             results.Add(element);
1534         }
1535     }
1536 }
1537 }
1538 }
1539
1540 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1541 {
1542     var result = new ulong[5];
1543     TryStepRight(startLink, rightLink, result, 0);
1544     Links.Each(_constants.Any, startLink, couple =>
1545     {
1546         if (couple != startLink)
1547         {
1548             if (TryStepRight(couple, rightLink, result, 2))
1549             {
1550                 return false;
1551             }
1552         }
1553         return true;
1554     });
1555     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1556     {
1557         result[4] = startLink;
1558     }
1559     return result;

```

```

1559     }
1560
1561     public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1562     {
1563         var added = 0;
1564         Links.Each(startLink, _constants.Any, couple =>
1565         {
1566             if (couple != startLink)
1567             {
1568                 var coupleTarget = Links.GetTarget(couple);
1569                 if (coupleTarget == rightLink)
1570                 {
1571                     result[offset] = couple;
1572                     if (++added == 2)
1573                     {
1574                         return false;
1575                     }
1576                 }
1577                 else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1578                     ↳ == Net.And &&
1579                 {
1580                     result[offset + 1] = couple;
1581                     if (++added == 2)
1582                     {
1583                         return false;
1584                     }
1585                 }
1586             }
1587             return true;
1588         });
1589         return added > 0;
1590     }
1591
1592     public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1593     {
1594         var result = new ulong[5];
1595         TryStepLeft(startLink, leftLink, result, 0);
1596         Links.Each(startLink, _constants.Any, couple =>
1597         {
1598             if (couple != startLink)
1599             {
1600                 if (TryStepLeft(couple, leftLink, result, 2))
1601                 {
1602                     return false;
1603                 }
1604             }
1605             return true;
1606         });
1607         if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1608         {
1609             result[4] = leftLink;
1610         }
1611         return result;
1612     }
1613
1614     public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1615     {
1616         var added = 0;
1617         Links.Each(_constants.Any, startLink, couple =>
1618         {
1619             if (couple != startLink)
1620             {
1621                 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             }
1639             return true;
1640         });
1641         return added > 0;
1642     }
1643
1644 #endregion
1645
1646 #region Walkers
1647
1648 public class PatternMatcher : RightSequenceWalker<ulong>
1649 {
1650     private readonly Sequences _sequences;
1651     private readonly ulong[] _patternSequence;
1652     private readonly HashSet<LinkIndex> _linksInSequence;
1653     private readonly HashSet<LinkIndex> _results;
1654
1655     #region Pattern Match
1656
1657     enum PatternBlockType
1658     {
1659         Undefined,
1660         Gap,
1661         Elements
1662     }
1663
1664     struct PatternBlock
1665     {
1666         public PatternBlockType Type;
1667         public long Start;
1668         public long Stop;
1669     }
1670
1671     private readonly List<PatternBlock> _pattern;
1672     private int _patternPosition;
1673     private long _sequencePosition;
1674
1675 #endregion
1676
1677     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1678         ↳ HashSet<LinkIndex> results)
1679         : base(sequences.Links.Unsync)
1680     {
1681         _sequences = sequences;
1682         _patternSequence = patternSequence;
1683         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x
1684             ↳ != _constants.Any && x != ZeroOrMany));

```

```

1682     _results = results;
1683     _pattern = CreateDetailedPattern();
1684 }
1685
1686 protected override bool IsElement(IList<ulong> link) =>
    ↳ _linksInSequence.Contains(Links.GetIndex(link)) || base.IsElement(link);
1687
1688 public bool PatternMatch(LinkIndex sequenceToMatch)
1689 {
1690     _patternPosition = 0;
1691     sequencePosition = 0;
1692     foreach (var part in Walk(sequenceToMatch))
1693     {
1694         if (!PatternMatchCore(Links.GetIndex(part)))
1695         {
1696             break;
1697         }
1698     }
1699     return _patternPosition == _pattern.Count || (_patternPosition ==
    ↳ _pattern.Count - 1 && _pattern[_patternPosition].Start == 0);
1700 }
1701
1702 private List<PatternBlock> CreateDetailedPattern()
1703 {
1704     var pattern = new List<PatternBlock>();
1705     var patternBlock = new PatternBlock();
1706     for (var i = 0; i < _patternSequence.Length; i++)
1707     {
1708         if (patternBlock.Type == PatternBlockType.Undefined)
1709         {
1710             if (_patternSequence[i] == _constants.Any)
1711             {
1712                 patternBlock.Type = PatternBlockType.Gap;
1713                 patternBlock.Start = 1;
1714                 patternBlock.Stop = 1;
1715             }
1716             else if (_patternSequence[i] == ZeroOrMany)
1717             {
1718                 patternBlock.Type = PatternBlockType.Gap;
1719                 patternBlock.Start = 0;
1720                 patternBlock.Stop = long.MaxValue;
1721             }
1722             else
1723             {
1724                 patternBlock.Type = PatternBlockType.Elements;
1725                 patternBlock.Start = i;
1726                 patternBlock.Stop = i;
1727             }
1728         }
1729         else if (patternBlock.Type == PatternBlockType.Elements)
1730         {
1731             if (_patternSequence[i] == _constants.Any)
1732             {
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     }
1757     else // patternBlock.Type == PatternBlockType.Gap
1758     {
1759         if (_patternSequence[i] == _constants.Any)
1760         {
1761             patternBlock.Start++;
1762             if (patternBlock.Stop < patternBlock.Start)
1763             {
1764                 patternBlock.Stop = patternBlock.Start;
1765             }
1766         }
1767         else if (_patternSequence[i] == ZeroOrMany)
1768         {
1769             patternBlock.Stop = long.MaxValue;
1770         }
1771         else
1772         {
1773             pattern.Add(patternBlock);
1774             patternBlock = new PatternBlock
1775             {
1776                 Type = PatternBlockType.Elements,
1777                 Start = i,
1778                 Stop = i
1779             };
1780         }
1781     }
1782     if (patternBlock.Type != PatternBlockType.Undefined)
1783     {
1784         pattern.Add(patternBlock);
1785     }
1786     return pattern;
1787 }
1788
1789 ///  

1790 // * match: search for regexp anywhere in text */  

1791 int match(char* regexp, char* text)  

1792 {  

1793     do  

1794     {  

1795         while (*text++ != '\0');  

1796         return 0;  

1797     }  

1798 }  

1799  

1800 ///  

1801 // * matchhere: search for regexp at beginning of text */  

1802 int matchhere(char* regexp, char* text)  

1803 {  

1804     if (regexp[0] == '\0')  

1805         return 1;  

1806     if (regexp[1] == '*')  

1807         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;
1822 //}
1823
1824 //private void GetNextPatternElement(out LinkIndex element, out long
1825 //    ↪ mininumGap, out long maximumGap)
1826 //{
1827 //    mininumGap = 0;
1828 //    maximumGap = 0;
1829 //    element = 0;
1830 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1831 //    {
1832 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1833 //            mininumGap++;
1834 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1835 //            maximumGap = long.MaxValue;
1836 //        else
1837 //            break;
1838 //    }
1839 //    if (maximumGap < mininumGap)
1840 //        maximumGap = mininumGap;
1841 //}
1842
1843 private bool PatternMatchCore(LinkIndex element)
1844 {
1845     if (_patternPosition >= _pattern.Count)
1846     {
1847         _patternPosition = -2;
1848         return false;
1849     }
1850     var currentPatternBlock = _pattern[_patternPosition];
1851     if (currentPatternBlock.Type == PatternBlockType.Gap)
1852     {
1853         //var currentMatchingBlockLength = (_sequencePosition -
1854         ↪ _lastMatchedBlockPosition);
1855         if (_sequencePosition < currentPatternBlock.Start)
1856         {
1857             _sequencePosition++;
1858             return true; // Двигаемся дальше
1859         }
1860         // Это последний блок
1861         if (_pattern.Count == _patternPosition + 1)
1862         {
1863             _patternPosition++;
1864             _sequencePosition = 0;
1865             return false; // Полное соответствие
1866         }
1867     }

```

```

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 #endregion
1953 }
1954 }
1955 }

```

## ./Sequences/Sequences.Experiments.ReadSequence.cs

```

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

```

```

32     #endif
33     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,
9              ↳ IList<TLink>> groupedSequence)
10         {
11             var finalSequence = new TLink[groupedSequence.Count];
12             for (var i = 0; i < finalSequence.Length; i++)
13             {
14                 var part = groupedSequence[i];
15                 finalSequence[i] = part.Length == 1 ? part[0] : sequences.Create(part);
16             }
17             return sequences.Create(finalSequence);
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
22         /// ранее.
23         ↳

```

```

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

```

```

76     var indexed = true;
77     var i = sequence.Count;
78     while (--i >= 1 && (indexed =
    ↪     !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1],
    ↪     sequence[i]), _null))) { }
79     return indexed;
80 }
81 }
82 }

```

## ./Sequences/SequencesOptions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
5  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
6  using Platform.Data.Doublets.Sequences.Converters;
7  using Platform.Data.Doublets.Sequences.CriteriaMatchers;
8
9  namespace Platform.Data.Doublets.Sequences
10 {
11     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
    ↪     ILinks<TLink> must contain GetConstants function.
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪     EqualityComparer<TLink>.Default;
14
15         public TLink SequenceMarkerLink { get; set; }
16         public bool UseCascadeUpdate { get; set; }
17         public bool UseCascadeDelete { get; set; }
18         public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
19         public bool UseSequenceMarker { get; set; }
20         public bool UseCompression { get; set; }
21         public bool UseGarbageCollection { get; set; }
22         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
23         public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
24
25         public MarkedSequenceCriteriaMatcher<TLink> MarkedSequenceMatcher { get; set; }
26         public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
27         public SequencesIndexer<TLink> Indexer { get; set; }
28
29         // TODO: Реализовать компактификацию при чтении
30         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
31         //public bool UseRequestMarker { get; set; }
32         //public bool StoreRequestResults { get; set; }
33
34         public void InitOptions(ISynchronizedLinks<TLink> links)
35         {
36             if (UseSequenceMarker)
37             {
38                 if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
39                 {
40                     SequenceMarkerLink = links.CreatePoint();
41                 }
42             }
43             else
44             {
45                 if (!links.Exists(SequenceMarkerLink))
46                 {
47                     var link = links.CreatePoint();
48                     if (!_equalityComparer.Equals(link, SequenceMarkerLink))
49                     {

```

```

        throw new InvalidOperationException("Cannot recreate sequence
        ↪     marker link.");
    }
}

```

```

50     }
51 }
52 }
53 if (MarkedSequenceMatcher == null)
54 {
55     MarkedSequenceMatcher = new
    ↪     MarkedSequenceCriteriaMatcher<TLink>(links, SequenceMarkerLink);
56 }
57 }
58 var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
59 if (UseCompression)
60 {
61     if (LinksToSequenceConverter == null)
62     {
63         ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
64         if (UseSequenceMarker)
65         {
66             totalSequenceSymbolFrequencyCounter = new
    ↪             TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
    ↪             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
45                     ↪ amount of NIL characters before actual Character)
46                     var createdLink = _links.CreatePoint();
47                     _links.Update(createdLink, firstLink, createdLink);
48                     if (createdLink != i)
49                     {
50                         throw new InvalidOperationException("Unable to initialize UTF 16 table.");
51                     }
52                 }
53             }
54         }
55
56         // 0 - null link
57         // 1 - nil character (0 character)
58         // ...
59         // 65536 (0(1) + 65535 = 65536 possible values)
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         public static ulong FromCharToLink(char character) => (ulong)character + 1;
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         public static char FromLinkToChar(ulong link) => (char)(link - 1);

```

```

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             return sb.ToString();
93         }
94
95         public static ulong[] FromCharsToLinkArray(char[] chars) =>
96             ↪ FromCharsToLinkArray(chars, chars.Length);
97
98         public static ulong[] FromCharsToLinkArray(char[] chars, int count)
99         {
100             // char array to ulong array
101             var linksSequence = new ulong[count];
102             for (var i = 0; i < count; i++)
103             {
104                 linksSequence[i] = FromCharToLink(chars[i]);
105             }
106             return linksSequence;
107         }
108
109         public static ulong[] FromStringToLinkArray(string sequence)
110         {
111             // char array to ulong array
112             var linksSequence = new ulong[sequence.Length];
113             for (var i = 0; i < sequence.Length; i++)
114             {
115                 linksSequence[i] = FromCharToLink(sequence[i]);
116             }
117             return linksSequence;
118         }
119
120         public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
121         {
122             var result = new List<ulong[]>();
123             var offset = 0;
124             while (offset < sequence.Length)
125             {
126                 var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
127                 var relativeLength = 1;

```

```

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

```

```

184         return result;
185     }
186 }
187 }

```

## ./Sequences/Walkers/LeftSequenceWalker.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Data.Doublets.Sequences.Walkers
5  {
6      public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
7      {
8          public LeftSequenceWalker(ILinks<TLink> links) : base(links) { }
9
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         protected override IList<TLink> GetNextElementAfterPop(IList<TLink> element)
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>
20             ↪ element)
21         {
22             var start = Links.Constants.IndexPart + 1;
23             for (var i = element.Count - 1; i >= start; i--)
24             {
25                 var partLink = Links.GetLink(element[i]);
26                 if (IsElement(partLink))
27                 {
28                     yield return partLink;
29                 }
30             }
31         }
32     }
33 }

```

## ./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>
20             ↪ element)
21         {
22             var start = Links.Constants.IndexPart + 1;
23             for (var i = element.Count - 1; i >= start; i--)
24             {
25                 var partLink = Links.GetLink(element[i]);
26                 if (IsElement(partLink))
27                 {
28                     yield return partLink;
29                 }
30             }
31         }
32     }
33 }

```

```

19         for (var i = Links.Constants.IndexPart + 1; i < element.Count; i++)
20         {
21             var partLink = Links.GetLink(element[i]);
22             if (IsElement(partLink))
23             {
24                 yield return partLink;
25             }
26         }
27     }
28 }
29 }

```

## ./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
11         ↪ contain 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)
22                 ↪ _stack.Pop()
23             }
24             var element = Links.GetLink(sequence);
25             if (IsElement(element))
26             {
27                 yield return element;
28             }
29             else
30             {
31                 while (true)
32                 {
33                     if (IsElement(element))
34                     {
35                         if (_stack.Count == 0)
36                         {
37                             break;
38                         }
39                         element = _stack.Pop();
40                         foreach (var output in WalkContents(element))
41                         {
42                             yield return output;
43                         }
44                         element = GetNextElementAfterPop(element);
45                     }
46                     else
47                     {
48                         _stack.Push(element);
49                         element = GetNextElementAfterPush(element);
50                     }
51                 }
52             }
53         }
54     }
55 }

```

```

46     }
47 }
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>
63     ↪ element);
64 }
65 }

```

## ./Stacks/Stack.cs

```

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

## ./SynchronizedLinks.cs

```
1 using System;
2 using System.Collections.Generic;
3 using Platform.Data.Constants;
4 using Platform.Data.Doublets;
5 using Platform.Threading.Synchronization;
6
7 namespace Platform.Data.Doublets
8 {
9     /// <remarks>
10     /// TODO: Autogeneration of synchronized wrapper (decorator).
11     /// TODO: Try to unfold code of each method using IL generation for 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(), links) { }
22
23         public SynchronizedLinks(ISynchronization synchronization, ILinks<T> links)
24         {
25             SyncRoot = synchronization;
26             Sync = this;
27             Unsync = links;
28             Constants = links.Constants;
29         }
30
31         public T Count(IList<T> restriction) =>
        ↳ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
32         public T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
        ↳ SyncRoot.ExecuteReadOperation(handler, restrictions, (handler1, restrictions1)
        ↳ => Unsync.Each(handler1, restrictions1));
33         public T Create() => SyncRoot.ExecuteWriteOperation(Unsync.Create);
34         public T Update(IList<T> restrictions) =>
        ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Update);
35         public void Delete(T link) => SyncRoot.ExecuteWriteOperation(link, Unsync.Delete);
36     }
```

```
37         public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T>
        ↳ matchedHandler, IList<T> substitution, Func<IList<T>, IList<T>, T>
        ↳ substitutedHandler)
38         //{
39         //    if (restriction != null && substitution != null &&
        ↳ !substitution.EqualTo(restriction))
40         //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
        ↳ substitution, substitutedHandler, Unsync.Trigger);
41
42         //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler,
        ↳ substitution, substitutedHandler, Unsync.Trigger);
43         //}
44     }
45 }
```

## ./UInt64Link.cs

```
1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using Platform.Exceptions;
5 using Platform.Ranges;
6 using Platform.Helpers.Singletons;
7 using Platform.Data.Constants;
8
9 namespace Platform.Data.Doublets
10 {
11     /// <summary>
12     /// Структура описывающая уникальную связь.
13     /// </summary>
14     public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>,
        ↳ IList<ulong>
15     {
16         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
        ↳ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
17
18         private const int Length = 3;
19
20         public readonly ulong Index;
21         public readonly ulong Source;
22         public readonly ulong Target;
23
24         public static readonly UInt64Link Null = new UInt64Link();
25
26         public UInt64Link(params ulong[] values)
27         {
28             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
        ↳ _constants.Null;
29             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
        ↳ _constants.Null;
30             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
        ↳ _constants.Null;
31         }
32
33         public UInt64Link(IList<ulong> values)
34         {
35             Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
        ↳ _constants.Null;
36             Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
        ↳ _constants.Null;
37             Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
        ↳ _constants.Null;
38         }
39     }
```

```

40 public UInt64Link(ulong index, ulong source, ulong target)
41 {
42     Index = index;
43     Source = source;
44     Target = target;
45 }
46
47 public UInt64Link(ulong source, ulong target)
48 : this(_constants.Null, source, target)
49 {
50     Source = source;
51     Target = target;
52 }
53
54 public static UInt64Link Create(ulong source, ulong target) => new
55     ↪ UInt64Link(source, target);
56
57 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
58
59 public bool IsNull() => Index == _constants.Null
60     && Source == _constants.Null
61     && Target == _constants.Null;
62
63 public override bool Equals(object other) => other is UInt64Link &&
64     ↪ Equals((UInt64Link)other);
65
66 public bool Equals(UInt64Link other) => Index == other.Index
67     && Source == other.Source
68     && Target == other.Target;
69
70 public static string ToString(ulong index, ulong source, ulong target) => §"({index}:
71     ↪ {source}->{target})";
72
73 public static string ToString(ulong source, ulong target) => §"({source}->{target})";
74
75 public static implicit operator ulong[(UInt64Link link) => link.ToArray());
76
77 public static implicit operator UInt64Link(ulong[] linkArray) => new
78     ↪ UInt64Link(linkArray);
79
80 public ulong[] ToArray()
81 {
82     var array = new ulong[Length];
83     CopyTo(array, 0);
84     return array;
85 }
86
87 public override string ToString() => Index == _constants.Null ? ToString(Source,
88     ↪ Target) : ToString(Index, Source, Target);
89
90 #region IList
91
92 public ulong this[int index]
93 {
94     get
95     {
96         Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
97             ↪ nameof(index));
98         if (index == _constants.IndexPart)
99         {
100             return Index;
101         }
102         if (index == _constants.SourcePart)
103         {

```

```

98             return Source;
99         }
100         if (index == _constants.TargetPart)
101         {
102             return Target;
103         }
104         throw new NotSupportedException(); // Impossible path due to
105             ↪ Ensure.ArgumentInRange
106     }
107     set => throw new NotSupportedException();
108 }
109
110 public int Count => Length;
111
112 public bool IsReadOnly => true;
113
114 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
115
116 public IEnumerator<ulong> GetEnumerator()
117 {
118     yield return Index;
119     yield return Source;
120     yield return Target;
121 }
122
123 public void Add(ulong item) => throw new NotSupportedException();
124
125 public void Clear() => throw new NotSupportedException();
126
127 public bool Contains(ulong item) => IndexOf(item) >= 0;
128
129 public void CopyTo(ulong[] array, int arrayIndex)
130 {
131     Ensure.Always.ArgumentNotNull(array, nameof(array));
132     Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length -
133         ↪ 1), nameof(arrayIndex));
134     if (arrayIndex + Length > array.Length)
135     {
136         throw new ArgumentException();
137     }
138     array[arrayIndex++] = Index;
139     array[arrayIndex++] = Source;
140     array[arrayIndex] = Target;
141 }
142
143 public bool Remove(ulong item) =>
144     ↪ Throw.A.NotSupportedExceptionAndReturn<bool>();
145
146 public int IndexOf(ulong item)
147 {
148     if (Index == item)
149     {
150         return _constants.IndexPart;
151     }
152     if (Source == item)
153     {
154         return _constants.SourcePart;
155     }
156     if (Target == item)
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) =>
6             ↪ Point<ulong>.IsFullPoint(link);
7         public static bool IsPartialPoint(this UInt64Link link) =>
8             ↪ Point<ulong>.IsPartialPoint(link);
9     }
10 }

```

## ./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) =>
17             ↪ UnicodeMap.InitNew(links);
18
19         public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong>
20             ↪ sequence)
21         {
22             if (sequence == null)
23             {
24                 return;
25             }
26             for (var i = 0; i < sequence.Count; i++)
27             {
28                 if (!links.Exists(sequence[i]))
29                 {
30                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
31                         ↪ §"sequence[{i}]");
32                 }
33             }
34
35             public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links,
36                 ↪ IList<ulong> sequence)
37         {
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85

```

```

34         if (sequence == null)
35         {
36             return;
37         }
38         for (var i = 0; i < sequence.Count; i++)
39         {
40             if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
41             {
42                 throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
43                     ↪ §"sequence[{i}]");
44             }
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
65     public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
66         ↪ Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug
67         ↪ = false)
68     {
69         var sb = new StringBuilder();
70         var visited = new HashSet<ulong>();
71         links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
72             ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
73         return sb.ToString();
74     }
75
76     public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
77         ↪ Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link>
78         ↪ appendElement, bool renderIndex = false, bool renderDebug = false)
79     {
80         var sb = new StringBuilder();
81         var visited = new HashSet<ulong>();
82         links.AppendStructure(sb, visited, linkIndex, isElement, appendElement,
83             ↪ renderIndex, renderDebug);
84         return sb.ToString();
85     }
86
87     public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
88         ↪ HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
89         ↪ Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false,
90         ↪ bool renderDebug = false)
91     {
92         if (sb == null)
93         {
94             throw new ArgumentNullException(nameof(sb));
95         }
96     }

```

```

86     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
    ↪ Constants.Itself)
87     {
88         return;
89     }
90     if (links.Exists(linkIndex))
91     {
92         if (visited.Add(linkIndex))
93         {
94             sb.Append('(');
95             var link = new UInt64Link(links.GetLink(linkIndex));
96             if (renderIndex)
97             {
98                 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,
    ↪ appendElement, renderIndex);
115                 }
116             }
117             sb.Append(' ');
118             if (link.Target == link.Index)
119             {
120                 sb.Append(link.Index);
121             }
122             else
123             {
124                 var target = new UInt64Link(links.GetLink(link.Target));
125                 if (isElement(target))
126                 {
127                     appendElement(sb, target);
128                 }
129                 else
130                 {
131                     links.AppendStructure(sb, visited, target.Index, isElement,
    ↪ appendElement, renderIndex);
132                 }
133             }
134             sb.Append(')');
135         }
136         else
137         {
138             if (renderDebug)
139             {
140                 sb.Append('(' * 1);
141             }
142             sb.Append(linkIndex);
143         }
144     }

```

```

145     else
146     {
147         if (renderDebug)
148         {
149             sb.Append('~');
150         }
151         sb.Append(linkIndex);
152     }
153 }
154 }
155 }

```

# ./UInt64LinksTransactionsLayer.cs

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Runtime.CompilerServices;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Platform.Disposables;
9  using Platform.Timestamps;
10 using Platform.Unsafe;
11 using Platform.IO;
12 using Platform.Data.Doublets.Decorators;
13
14 namespace Platform.Data.Doublets
15 {
16     public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong>
17     {
18         ↪ // -V3073
19         // <remarks>
20         // Альтернативные варианты хранения трансформации (элемента транзакции):
21         // private enum TransitionType
22         // {
23         //     Creation,
24         //     UpdateOf,
25         //     UpdateTo,
26         //     Deletion
27         // }
28         // private struct Transition
29         // {
30         //     public ulong TransactionId;
31         //     public UniqueTimestamp Timestamp;
32         //     public TransactionItemType Type;
33         //     public Link Source;
34         //     public Link Linker;
35         //     public Link Target;
36         // }
37         // Или
38         // public struct TransitionHeader
39         // {
40         //     public ulong TransactionIdCombined;
41         //     public ulong TimestampCombined;
42         //     public ulong TransactionId
43         //     {
44         //         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
92     ↪ transactionId, 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
101     ↪ transactionId, UInt64Link before)
102     : this(uniqueTimestampFactory, transactionId, before, default)
103     {
104     }
105
106     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
107     ↪ transactionId)
108     : this(uniqueTimestampFactory, transactionId, default, default)
109     {
110     }
111 }

```

```

108     public override string ToString() => $"{Timestamp} {TransactionId}: {Before}
109     ↪ => {After}";
110 }
111
112 <remarks>
113 Другие варианты реализации транзакций (атомарности):
114 1. Разделение хранения значения связи ((Source Target) или (Source Linker
115     ↪ Target)) и индексов.
116 2. Хранение трансформаций/операций в отдельном хранилище Links, но
117     ↪ дополнительно потребуется решить вопрос
118     со ссылками на внешние идентификаторы, или как-то иначе решить
119     ↪ вопрос с пересечениями идентификаторов.
120
121 Где хранить промежуточный список транзакций?
122
123 В оперативной памяти:
124 Минусы:
125 1. Может усложнить систему, если она будет функционировать
126     ↪ самостоятельно,
127     так как нужно отдельно выделять память под список трансформаций.
128 2. Выделенной оперативной памяти может не хватить, в том случае,
129     ↪ если транзакция использует слишком много трансформаций.
130     -> Можно использовать жёсткий диск для слишком длинных транзакций.
131     -> Максимальный размер списка трансформаций можно ограничить /
132     задать константой.
133 3. При подтверждении транзакции (Commit) все трансформации
134     ↪ записываются разом создавая задержку.
135
136 На жёстком диске:
137 Минусы:
138 1. Длительный отклик, на запись каждой трансформации.
139 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
140     ↪ -> Это может решаться упаковкой/исключением дублирующих операций.
141     -> Также это может решаться тем, что короткие транзакции вообще
142     не будут записываться в случае отката.
143 3. Перед тем как выполнять отмену операций транзакции нужно дождаться
144     ↪ пока все операции (трансформации)
145     будут записаны в лог.
146
147 </remarks>
148 public class Transaction : DisposableBase
149 {
150     private readonly Queue<Transition> _transitions;
151     private readonly UInt64LinksTransactionsLayer _layer;
152     public bool IsCommitted { get; private set; }
153     public bool IsReverted { get; private set; }
154
155     public Transaction(UInt64LinksTransactionsLayer layer)
156     {
157         _layer = layer;
158         if (_layer._currentTransactionId != 0)
159         {
160             throw new NotSupportedException("Nested transactions not supported.");
161         }
162         IsCommitted = false;
163         IsReverted = false;
164         _transitions = new Queue<Transition>();
165         SetCurrentTransaction(layer, this);
166     }
167
168     public void Commit()

```

```

162     {
163         EnsureTransactionAllowsWriteOperations(this);
164         while (_transitions.Count > 0)
165         {
166             var transition = _transitions.Dequeue();
167             _layer._transitions.Enqueue(transition);
168         }
169         layer._lastCommittedTransactionId = _layer._currentTransactionId;
170         IsCommitted = true;
171     }
172
173     private void Revert()
174     {
175         EnsureTransactionAllowsWriteOperations(this);
176         var transitionsToRevert = new Transition[_transitions.Count];
177         _transitions.CopyTo(transitionsToRevert, 0);
178         for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
179         {
180             _layer.RevertTransition(transitionsToRevert[i]);
181         }
182         IsReverted = true;
183     }
184
185     public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
186         ↳ Transaction transaction)
187     {
188         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
189         layer._currentTransactionTransitions = transaction._transitions;
190         layer._currentTransaction = transaction;
191     }
192
193     public static void EnsureTransactionAllowsWriteOperations(Transaction
194         ↳ transaction)
195     {
196         if (transaction.IsReverted)
197         {
198             throw new InvalidOperationException("Transation is reverted.");
199         }
200         if (transaction.IsCommitted)
201         {
202             throw new InvalidOperationException("Transation is committed.");
203         }
204     }
205
206     protected override void DisposeCore(bool manual, bool wasDisposed)
207     {
208         if (!wasDisposed && _layer != null && !_layer.IsDisposed)
209         {
210             if (!IsCommitted && !IsReverted)
211             {
212                 Revert();
213             }
214             _layer.ResetCurrentTransaction();
215         }
216
217         // TODO: THIS IS EXCEPTION WORKAROUND, REMOVE IT THEN
218         ↳ https://github.com/linksplatform/Disposables/issues/13 FIXED
219         protected override bool AllowMultipleDisposeCalls => true;
220
221     }
222
223     public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);

```

```

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 =
245             ↳ FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
246         var lastWrittenTransition =
247             ↳ FileHelpers.ReadLastOrDefault<Transition>(logAddress);
248         if (!lastCommittedTransition.Equals(lastWrittenTransition))
249         {
250             Dispose();
251             throw new NotSupportedException("Database is damaged, autorecovery is not
252                 ↳ supported yet.");
253         }
254         if (lastCommittedTransition.Equals(default(Transition)))
255         {
256             FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
257         }
258         lastCommittedTransition = lastCommittedTransition;
259         // TODO: Think about a better way to calculate or store this value
260         var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
261         _lastCommittedTransactionId = allTransitions.Max(x => x.TransactionId);
262         _uniqueTimestampFactory = new UniqueTimestampFactory();
263         _logAddress = logAddress;
264         _log = FileHelpers.Append(logAddress);
265         _transitions = new Queue<Transition>();
266         _transitionsPusher = new Task(TransitionsPusher);
267         _transitionsPusher.Start();
268     }
269
270     public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
271
272     public override ulong Create()
273     {
274         var createdLinkIndex = Links.Create();
275         var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
276         CommitTransition(new Transition(_uniqueTimestampFactory,
277             ↳ _currentTransactionId, default, createdLink));
278         return createdLinkIndex;
279     }
280
281     public override ulong Update(IList<ulong> parts)
282     {
283         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, beforeLink, afterLink));
283     return parts[Constants.IndexPart];
284 }
285
286 public override void Delete(ulong link)
287 {
288     var deletedLink = new UInt64Link(Links.GetLink(link));
289     Links.Delete(link);
290     CommitTransition(new Transition(_uniqueTimestampFactory,
    ↪     _currentTransactionId, deletedLink, default));
291 }
292
293 [MethodImpl(MethodImplOptions.AggressiveInlining)]
294 private Queue<Transition> GetCurrentTransitions() =>
    ↪     _currentTransactionTransitions ?? _transitions;
295
296 private void CommitTransition(Transition transition)
297 {
298     if (_currentTransaction != null)
299     {
300         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
301     }
302     var transitions = GetCurrentTransitions();
303     transitions.Enqueue(transition);
304 }
305
306 private void RevertTransition(Transition transition)
307 {
308     if (transition.After.IsNull()) // Revert Deletion with Creation
309     {
310         Links.Create();
311     }
312     else if (transition.Before.IsNull()) // Revert Creation with Deletion
313     {
314         Links.Delete(transition.After.Index);
315     }
316     else // Revert Update
317     {
318         Links.Update(new[] { transition.After.Index, transition.Before.Source,
    ↪     transition.Before.Target });
319     }
320 }
321
322 private void ResetCurrentTransation()
323 {
324     _currentTransactionId = 0;
325     _currentTransactionTransitions = null;
326     _currentTransaction = null;
327 }
328
329 private void PushTransitions()
330 {
331     if (_log == null || _transitions == null)
332     {

```

```

333         return;
334     }
335     for (var i = 0; i < _transitions.Count; i++)
336     {
337         var transition = _transitions.Dequeue();
338
339         _log.Write(transition);
340         _lastCommittedTransition = transition;
341     }
342 }
343
344 private void TransitionsPusher()
345 {
346     while (!IsDisposed && _transitionsPusher != null)
347     {
348         Thread.Sleep(DefaultPushDelay);
349         PushTransitions();
350     }
351 }
352
353 public Transaction BeginTransaction() => new Transaction(this);
354
355 private void DisposeTransitions()
356 {
357     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, 1  
./Converters/UnaryNumberToAddressAddOperationConverter.cs, 2  
./Converters/UnaryNumberToAddressOrOperationConverter.cs, 2  
./Decorators/LinksCascadeDependenciesResolver.cs, 3  
./Decorators/LinksCascadeUniquenessAndDependenciesResolver.cs, 3  
./Decorators/LinksDecoratorBase.cs, 3  
./Decorators/LinksDependenciesValidator.cs, 4  
./Decorators/LinksDisposableDecoratorBase.cs, 4  
./Decorators/LinksInnerReferenceValidator.cs, 4  
./Decorators/LinksNonExistentReferencesCreator.cs, 4  
./Decorators/LinksNullToSelfReferenceResolver.cs, 5  
./Decorators/LinksSelfReferenceResolver.cs, 5  
./Decorators/LinksUniquenessResolver.cs, 5  
./Decorators/LinksUniquenessValidator.cs, 6  
./Decorators/NonNullContentsLinkDeletionResolver.cs, 6  
./Decorators/UInt64Links.cs, 6  
./Decorators/UniLinks.cs, 7  
./Doublet.cs, 10  
./DoubletComparer.cs, 10  
./Hybrid.cs, 10  
./ILinks.cs, 11  
./ILinksExtensions.cs, 11  
./ISynchronizedLinks.cs, 18  
./Incrementers/FrequencyIncrementer.cs, 17  
./Incrementers/LinkFrequencyIncrementer.cs, 17  
./Incrementers/UnaryNumberIncrementer.cs, 18  
./Link.cs, 18  
./LinkExtensions.cs, 20  
./LinksOperatorBase.cs, 20  
./PropertyOperators/DefaultLinkPropertyOperator.cs, 20  
./PropertyOperators/FrequencyPropertyOperator.cs, 20  
./ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs, 27  
./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs, 27  
./ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 21  
./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs, 35  
./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs, 36  
./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 31  
./Sequences/Converters/BalancedVariantConverter.cs, 40  
./Sequences/Converters/CompressingConverter.cs, 40  
./Sequences/Converters/LinksListToSequenceConverterBase.cs, 42  
./Sequences/Converters/OptimalVariantConverter.cs, 42  
./Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 43  
./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs, 44  
./Sequences/CreteriaMatchers/MarkedSequenceCreteriaMatcher.cs, 44  
./Sequences/DefaultSequenceAppender.cs, 44  
./Sequences/DuplicateSegmentsCounter.cs, 44  
./Sequences/DuplicateSegmentsProvider.cs, 45  
./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs, 46  
./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToltsFrequencyNumberConverter.cs, 46  
./Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 46  
./Sequences/Frequencies/Cache/LinkFrequency.cs, 47  
./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 48  
./Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 48  
./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 48  
./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 48  
./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 49  
./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 49  
./Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 49  
./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 50  
./Sequences/HeightProviders/ISequenceHeightProvider.cs, 50  
./Sequences/Sequences.Experiments.ReadSequence.cs, 73  
./Sequences/Sequences.Experiments.cs, 56  
./Sequences/Sequences.cs, 50  
./Sequences/SequencesExtensions.cs, 74  
./Sequences/SequencesIndexer.cs, 74  
./Sequences/SequencesOptions.cs, 75  
./Sequences/UnicodeMap.cs, 75  
./Sequences/Walkers/LeftSequenceWalker.cs, 77  
./Sequences/Walkers/RightSequenceWalker.cs, 77  
./Sequences/Walkers/SequenceWalkerBase.cs, 78  
./Stacks/Stack.cs, 78  
./Stacks/StackExtensions.cs, 79  
./SynchronizedLinks.cs, 79  
./UInt64Link.cs, 79  
./UInt64LinkExtensions.cs, 81  
./UInt64LinksExtensions.cs, 81  
./UInt64LinksTransactionsLayer.cs, 82  
./obj/Debug/netstandard2.0/Platform.Data.Doublets.AssemblyInfo.cs, 20