

LinksPlatform's Platform.Data.Doublets Class Library

./Converters/AddressToUnaryNumberConverter.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Reflection;
4  using Platform.Numbers;
5
6  namespace Platform.Data.Doublets.Converters
7  {
8      public class AddressToUnaryNumberConverter<TLink> :
9          ↳ LinksOperatorBase<TLink>, IConverter<TLink>
10     {
11         private static readonly EqualityComparer<TLink>
12             ↳ _equalityComparer = EqualityComparer<TLink>.Default;
13
14         private readonly IConverter<int, TLink>
15             ↳ _powerOf2ToUnaryNumberConverter;
16
17         public AddressToUnaryNumberConverter(ILinks<TLink> links,
18             ↳ IConverter<int, TLink> powerOf2ToUnaryNumberConverter) :
19             ↳ base(links) => _powerOf2ToUnaryNumberConverter =
20             ↳ powerOf2ToUnaryNumberConverter;
21
22         public TLink Convert(TLink sourceAddress)
23         {
24             var number = sourceAddress;
25             var target = Links.Constants.Null;
26             for (int i = 0; i < CachedTypeInfo<TLink>.BitsLength; i++)
27             {
28                 if (_equalityComparer.Equals(ArithmeticHelpers.And(num
29                     ↳ ber, Integer<TLink>.One),
30                     ↳ Integer<TLink>.One))
31                 {
32                     target = _equalityComparer.Equals(target,
33                         ↳ Links.Constants.Null)
34                     ? _powerOf2ToUnaryNumberConverter.Convert(i)
35                     : Links.GetOrCreate(_powerOf2ToUnaryNumberConv
36                         ↳ erter.Convert(i),
37                         ↳ target);
38                 }
39                 number =
40                     (Integer<TLink>)((ulong)(Integer<TLink>)number >>
41                         ↳ 1); // Should be BitwiseHelpers.ShiftRight(number,
42                         ↳ 1);
43                 if (_equalityComparer.Equals(number, default))
44                 {
45                     break;
46                 }
47             }
48             return target;
49         }
50     }
51 }

```

./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>
11             ↳ _equalityComparer = EqualityComparer<TLink>.Default;
12
13         private readonly ISpecificPropertyOperator<TLink, TLink>
14             ↳ _frequencyPropertyOperator;
15         private readonly IConverter<TLink>
16             ↳ _unaryNumberToAddressConverter;
17
18         public LinkToItsFrequencyNumberConveter(
19             ILinks<TLink> links,
20             ISpecificPropertyOperator<TLink, TLink>
21             ↳ frequencyPropertyOperator,
22             IConverter<TLink> unaryNumberToAddressConverter)
23             : base(links)
24         {
25             _frequencyPropertyOperator = frequencyPropertyOperator;
26             _unaryNumberToAddressConverter =
27                 ↳ unaryNumberToAddressConverter;
28         }
29
30         public TLink Convert(Doublet<TLink> doublet)
31         {
32             var link = Links.SearchOrDefault(doublet.Source,
33                 ↳ doublet.Target);
34             if (_equalityComparer.Equals(link, Links.Constants.Null))
35             {
36                 throw new ArgumentException($"Link with
37                     ↳ {doublet.Source} source and {doublet.Target}
38                     ↳ target not found.", nameof(doublet));
39             }
40             var frequency = _frequencyPropertyOperator.Get(link);
41             if (_equalityComparer.Equals(frequency, default))
42             {
43                 return default;
44             }
45             var frequencyNumber = Links.GetSource(frequency);
46             var number = _unaryNumberToAddressConverter.Convert(freque
47                 ↳ ncyNumber);
48             return number;
49         }
50     }
51 }

```

./Converters/PowerOf2ToUnaryNumberConverter.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Converters
6  {
7      public class PowerOf2ToUnaryNumberConverter<TLink> :
8          ↳ LinksOperatorBase<TLink>, IConverter<int, TLink>
9      {

```

```

9     private static readonly EqualityComparer<TLink>
    ↪     _equalityComparer = EqualityComparer<TLink>.Default;
10
11     private readonly TLink[] _unaryNumberPowersOf2;
12
13     public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links,
    ↪     TLink one) : base(links)
14     {
15         _unaryNumberPowersOf2 = new TLink[64];
16         _unaryNumberPowersOf2[0] = one;
17     }
18
19     public TLink Convert(int power)
20     {
21         if (power < 0 || power >= _unaryNumberPowersOf2.Length)
22         {
23             throw new ArgumentOutOfRangeException(nameof(power));
24         }
25         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> :
    ↪      LinksOperatorBase<TLink>, IConverter<TLink>
8      {
9          private static readonly EqualityComparer<TLink>
    ↪          _equalityComparer = EqualityComparer<TLink>.Default;
10
11          private Dictionary<TLink, TLink> _unaryToUInt64;
12          private readonly TLink _unaryOne;
13
14          public UnaryNumberToAddressAddOperationConverter(ILinks<TLink>
    ↪          links, TLink unaryOne)
15              : base(links)
16          {
17              _unaryOne = unaryOne;
18              InitUnaryToUInt64();
19          }
20
21          private void InitUnaryToUInt64()
22          {
23              _unaryToUInt64 = new Dictionary<TLink, TLink>
24              {

```

```

25              { _unaryOne, Integer<TLink>.One }
26          };
27          var unary = _unaryOne;
28          var number = Integer<TLink>.One;
29          for (var i = 1; i < 64; i++)
30          {
31              _unaryToUInt64.Add(unary = Links.GetOrCreate(unary,
    ↪              ↪          unary), number =
    ↪              ↪          (Integer<TLink>)((Integer<TLink>)number * 2UL));
32          }
33
34      }
35
36      public TLink Convert(TLink unaryNumber)
37      {
38          if (_equalityComparer.Equals(unaryNumber, default))
39          {
40              return default;
41          }
42          if (_equalityComparer.Equals(unaryNumber, _unaryOne))
43          {
44              return Integer<TLink>.One;
45          }
46          var source = Links.GetSource(unaryNumber);
47          var target = Links.GetTarget(unaryNumber);
48          if (_equalityComparer.Equals(source, target))
49          {
50              return _unaryToUInt64[unaryNumber];
51          }
52          else
53          {
54              var result = _unaryToUInt64[source];
55              TLink lastValue;
56              while (!_unaryToUInt64.TryGetValue(target, out
    ↪              ↪          lastValue))
57              {
58                  source = Links.GetSource(target);
59                  result = ArithmeticHelpers.Add(result,
    ↪                  ↪          _unaryToUInt64[source]);
60                  target = Links.GetTarget(target);
61              }
62              result = ArithmeticHelpers.Add(result, lastValue);
63              return result;
64          }
65      }
66  }

```

./Converters/UnaryNumberToAddressOrOperationConverter.cs

```

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

```

```

11     private readonly IDictionary<TLink, int>
12     ↪ _unaryNumberPowerOf2Indicies;
13
14     public UnaryNumberToAddressOrOperationConverter(ILinks<TLink>
15     ↪ links, IConverter<int, TLink>
16     ↪ powerOf2ToUnaryNumberConverter)
17     : base(links)
18     {
19         _unaryNumberPowerOf2Indicies = new Dictionary<TLink,
20         ↪ int>();
21         for (int i = 0; i < CachedTypeInfo<TLink>.BitsLength; i++)
22         {
23             _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumber
24             ↪ Converter.Convert(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,
34         ↪ Links.Constants.Null))
35         {
36             if (_unaryNumberPowerOf2Indicies.TryGetValue(source,
37             ↪ out int powerOf2Index))
38             {
39                 source = Links.Constants.Null;
40             }
41             else
42             {
43                 powerOf2Index = _unaryNumberPowerOf2Indicies[Links
44                 ↪ .GetSource(source)];
45                 source = Links.GetTarget(source);
46             }
47             target = (Integer<TLink>)((Integer<TLink>)target | 1UL
48             ↪ << powerOf2Index); // MathHelpers.Or(target,
49             ↪ MathHelpers.ShiftLeft(One, powerOf2Index))
50         }
51         return target;
52     }
53 }

```

./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> :
8     ↪ LinksDecoratorBase<TLink>
9     {
10         private static readonly EqualityComparer<TLink>
11         ↪ _equalityComparer = EqualityComparer<TLink>.Default;

```

```

11     public LinksCascadeDependenciesResolver(ILinks<TLink> links) :
12     ↪ 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 =
23         ↪ (Integer<TLink>)Links.Count(Constants.Any, link);
24         var references =
25         ↪ ArrayPool.Allocate<TLink>((long)referencesCount);
26         var referencesFiller = new ArrayFiller<TLink,
27         ↪ TLink>(references, Constants.Continue);
28         Links.Each(referencesFiller.AddFirstAndReturnConstant,
29         ↪ Constants.Any, link);
30         //references.Sort() // TODO: Решить необходимо ли для
31         ↪ корректного порядка отмены операций в транзакциях
32         for (var i = (long)referencesCount - 1; i >= 0; i--)
33         {
34             if (!_equalityComparer.Equals(references[i], link))
35             {
36                 continue;
37             }
38             Links.Delete(references[i]);
39         }
40         ArrayPool.Free(references);
41     }
42 }

```

./Decorators/LinksCascadeUniquenessAndDependenciesResolver.cs

```

1 using System.Collections.Generic;
2 using Platform.Collections.Arrays;
3 using Platform.Numbers;
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public class LinksCascadeUniquenessAndDependenciesResolver<TLink>
8     ↪ : LinksUniquenessResolver<TLink>
9     {
10         private static readonly EqualityComparer<TLink>
11         ↪ _equalityComparer = EqualityComparer<TLink>.Default;
12
13         public LinksCascadeUniquenessAndDependenciesResolver(ILinks<TL
14         ↪ ink> links) : base(links) { }
15
16         protected override TLink ResolveAddressChangeConflict(TLink
17         ↪ oldLinkAddress, TLink newLinkAddress)
18         {
19             // TODO: Very similar to Merge (logic should be reused)
20             ulong referencesAsSourceCount =
21             ↪ (Integer<TLink>)Links.Count(Constants.Any,
22             ↪ oldLinkAddress, Constants.Any);

```

```

17     ulong referencesAsTargetCount =
18         ↳ (Integer<TLink>)Links.Count(Constants.Any,
19         ↳ Constants.Any, oldLinkAddress);
20     var references = ArrayPool.Allocate<TLink>((long)(referenc
21         ↳ esAsSourceCount +
22         ↳ referencesAsTargetCount));
23     var referencesFiller = new ArrayFiller<TLink,
24         ↳ TLink>(references, Constants.Continue);
25     Links.Each(referencesFiller.AddFirstAndReturnConstant,
26         ↳ Constants.Any, oldLinkAddress, Constants.Any);
27     Links.Each(referencesFiller.AddFirstAndReturnConstant,
28         ↳ Constants.Any, Constants.Any, oldLinkAddress);
29     for (ulong i = 0; i < referencesAsSourceCount; i++)
30     {
31         var reference = references[i];
32         if (!_equalityComparer.Equals(reference,
33         ↳ oldLinkAddress))
34         {
35             Links.Update(reference, newLinkAddress,
36                 ↳ Links.GetTarget(reference));
37         }
38     }
39     for (var i = (long)referencesAsSourceCount; i <
40         ↳ references.Length; i++)
41     {
42         var reference = references[i];
43         if (!_equalityComparer.Equals(reference,
44         ↳ oldLinkAddress))
45         {
46             Links.Update(reference,
47                 ↳ Links.GetSource(reference), newLinkAddress);
48         }
49     }
50     ArrayPool.Free(references);
51     return base.ResolveAddressChangeConflict(oldLinkAddress,
52         ↳ newLinkAddress);
53 }
54 }

```

./Decorators/LinksDecoratorBase.cs

```

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

```

```

19     public virtual T Count(IList<T> restriction) =>
20         ↳ Links.Count(restriction);
21
22     public virtual T Each(Func<IList<T>, T> handler, IList<T>
23         ↳ restrictions) => Links.Each(handler, restrictions);
24
25     public virtual T Create() => Links.Create();
26
27     public virtual T Update(IList<T> restrictions) =>
28         ↳ Links.Update(restrictions);
29
30     public virtual void Delete(T link) => Links.Delete(link);
31 }

```

./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) :
8             ↳ base(links) { }
9
10        public override T Update(IList<T> restrictions)
11        {
12            Links.EnsureNoDependencies(restrictions[Constants.IndexPar
13                ↳ t]);
14            return base.Update(restrictions);
15        }
16
17        public override void Delete(T link)
18        {
19            Links.EnsureNoDependencies(link);
20            base.Delete(link);
21        }
22    }
23 }

```

./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> :
9         ↳ DisposableBase, ILinks<T>
10     {
11         public LinksCombinedConstants<T, T, int> Constants { get; }
12
13         public readonly ILinks<T> Links;
14
15         protected LinksDisposableDecoratorBase(ILinks<T> links)
16         {
17             Links = links;
18             Constants = links.Constants;
19         }
20     }

```

```

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

```

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

```

```

30     public override void Delete(T link)
31     {
32         // TODO: Решить считать ли такое исключением, или лишь
33         ↳ более конкретным требованием?
34         Links.EnsureLinkExists(link, nameof(link));
35         base.Delete(link);
36     }
37 }

```

./Decorators/LinksNonExistentReferencesCreator.cs

```

1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      /// <remarks>
6      /// Not practical if newSource and newTarget are too big.
7      /// To be able to use practical version we should allow to create
8      ↳ link at any specific location inside
9      ↳ ResizableDirectMemoryLinks.
10     /// This in turn will require to implement not a list of empty
11     ↳ links, but a list of ranges to store it more efficiently.
12     /// </remarks>
13     public class LinksNonExistentReferencesCreator<T> :
14         ↳ LinksDecoratorBase<T>
15     {
16         public LinksNonExistentReferencesCreator(ILinks<T> links) :
17             ↳ base(links) { }
18
19         public override T Update(IList<T> restrictions)
20         {
21             Links.EnsureCreated(restrictions[Constants.SourcePart],
22                 ↳ restrictions[Constants.TargetPart]);
23             return base.Update(restrictions);
24         }
25     }
26 }

```

./Decorators/LinksNullToSelfReferenceResolver.cs

```

1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      public class LinksNullToSelfReferenceResolver<TLink> :
6         ↳ LinksDecoratorBase<TLink>
7     {
8         private static readonly EqualityComparer<TLink>
9             ↳ _equalityComparer = EqualityComparer<TLink>.Default;
10
11         public LinksNullToSelfReferenceResolver(ILinks<TLink> links) :
12             ↳ base(links) { }
13
14         public override TLink Create()
15         {
16             var link = base.Create();
17             return Links.Update(link, link, link);
18         }
19     }
20 }

```

```

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

```

```

27         restrictions[Constants.TargetPart] = _equalityComparer.Equ
            ↪ als(restrictions[Constants.TargetPart],
            ↪ Constants.Itself) ? restrictions[Constants.IndexPart]
            ↪ : restrictions[Constants.TargetPart];
28         return base.Update(restrictions);
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[Co
            ↪ nstants.SourcePart],
            ↪ restrictions[Constants.TargetPart]);
            if (_equalityComparer.Equals(newLinkAddress, default))
14             {
15                 return base.Update(restrictions);
16             }
17             return ResolveAddressChangeConflict(restrictions[Constants
            ↪ .IndexPart],
            ↪ newLinkAddress);
18         }
19
20         protected virtual TLink ResolveAddressChangeConflict(TLink
            ↪ oldLinkAddress, TLink newLinkAddress)
21         {
22             if (Links.Exists(oldLinkAddress))
23             {
24                 Delete(oldLinkAddress);
25             }
26             return newLinkAddress;
27         }
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]
            ↳ t],
            ↳ 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 расчитать 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 DefaultUniLinksBase, that contains logic
        ↳ itself and can be implemented using both IDoubletLinks and
        ↳ ILinks.)
19     /// </remarks>
20     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>,
        ↳ IUniLinks<TLink>
21     {
22         private static readonly EqualityComparer<TLink>
        ↳ _equalityComparer = EqualityComparer<TLink>.Default;
23
24         public UniLinks(ILinks<TLink> links) : base(links) { }
25
26         private struct Transition
27         {
28             public IList<TLink> Before;
29             public IList<TLink> After;
30
31             public Transition(IList<TLink> before, IList<TLink> after)
32             {
33                 Before = before;
34                 After = after;
35             }
36         }
37
38         public static readonly TLink NullConstant =
        ↳ Use<LinksCombinedConstants<TLink, TLink, int>>.Single.Null;
39         public static readonly IReadOnlyList<TLink> NullLink = new
        ↳ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant,
        ↳ NullConstant, NullConstant });
40
41         // TODO: Подумать о том, как реализовать древовидный
        ↳ Restriction и Substitution (Links-Expression)
42         public TLink Trigger(IList<TLink> restriction,
        ↳ Func<IList<TLink>, IList<TLink>, TLink> matchedHandler,
        ↳ IList<TLink> substitution, Func<IList<TLink>,
        ↳ IList<TLink>, TLink> substitutedHandler)
43         {
44             ///List<Transition> transitions = null;
45             ///if (!restriction.IsNullOrEmpty())
46             ///{
47                 /// // Есть причина делать проход (чтение)
48                 /// if (matchedHandler != null)
49                 /// {
50                     /// if (!substitution.IsNullOrEmpty())

```



```

137 //      Memory.FreeLink(restriction[Constants.IndexPart]);
138 //}
139 //else if (restriction.EqualTo(substitution)) // Read or
140 //    ("repeat" the state) // Each
141 //{
142 //    // No need to collect links to list
143 //    // Skip == Continue
144 //    // No need to check substitutedHandler
145 //    if (!Memory.Each(link =>
146 //        !Equals(matchedHandler(Memory.GetLinkValue(link)),
147 //        Constants.Break), restriction))
148 //        return Constants.Break;
149 //}
150 //else // Update
151 //{
152 //    //List<IList<T>> matchedLinks = null;
153 //    if (matchedHandler != null)
154 //    {
155 //        matchedLinks = new List<IList<T>>();
156 //        Func<T, bool> handler = link =>
157 //        {
158 //            var matchedLink = Memory.GetLinkValue(link);
159 //            var matchDecision =
160 //            matchedHandler(matchedLink);
161 //            if (Equals(matchDecision, Constants.Break))
162 //                return false;
163 //            if (!Equals(matchDecision, Constants.Skip))
164 //                matchedLinks.Add(matchedLink);
165 //            return true;
166 //        };
167 //        if (!Memory.Each(handler, restriction))
168 //            return Constants.Break;
169 //    }
170 //    if (!matchedLinks.IsNullOrEmpty())
171 //    {
172 //        var totalMatchedLinks = matchedLinks.Count;
173 //        for (var i = 0; i < totalMatchedLinks; i++)
174 //        {
175 //            var matchedLink = matchedLinks[i];
176 //            if (substitutedHandler != null)
177 //            {
178 //                var newValue = new List<T>(); // TODO:
179 //                Prepare value to update here
180 //                // TODO: Decide is it actually needed to
181 //                use Before and After substitution handling.
182 //                var substitutedDecision =
183 //                substitutedHandler(matchedLink, newValue);
184 //                if (Equals(substitutedDecision,
185 //                    Constants.Break))
186 //                    return Constants.Break;
187 //                if (Equals(substitutedDecision,
188 //                    Constants.Continue))
189 //                {
190 //                    // Actual update here
191 //                    Memory.SetLinkValue(newValue);
192 //                }
193 //                if (Equals(substitutedDecision,
194 //                    Constants.Skip))

```

```

185 //      {
186 //          // Cancel the update. TODO: decide
187 //          use separate Cancel constant or Skip is enough?
188 //      }
189 //    }
190 //  }
191 //}
192 return Constants.Continue;
193 }
194
195 public TLink Trigger(IList<TLink> patternOrCondition,
196 //    Func<IList<TLink>, TLink> matchHandler, IList<TLink>
197 //    substitution, Func<IList<TLink>, IList<TLink>, TLink>
198 //    substitutionHandler)
199 {
200     if (patternOrCondition.IsNullOrEmpty() &&
201         substitution.IsNullOrEmpty())
202     {
203         return Constants.Continue;
204     }
205     else if (patternOrCondition.EqualTo(substitution)) //
206         Should be Each here TODO: Check if it is a correct
207         condition
208     {
209         // Or it only applies to trigger without matchHandler.
210         throw new NotImplementedException();
211     }
212     else if (!substitution.IsNullOrEmpty()) // Creation
213     {
214         var before = ArrayPool<TLink>.Empty;
215         // Что должно означать False здесь? Остановиться
216         // (перестать идти) или пропустить (пройти мимо) или
217         // пустить (взять)?
218         if (matchHandler != null &&
219             _equalityComparer.Equals(matchHandler(before),
220             Constants.Break))
221         {
222             return Constants.Break;
223         }
224         var after = (IList<TLink>)substitution.ToArray();
225         if (_equalityComparer.Equals(after[0], default))
226         {
227             var newLink = Links.Create();
228             after[0] = newLink;
229         }
230         if (substitution.Count == 1)
231         {
232             after = Links.GetLink(substitution[0]);
233         }
234         else if (substitution.Count == 3)
235         {
236             Links.Update(after);
237         }
238         else
239         {
240             throw new NotSupportedException();
241         }
242         if (matchHandler != null)

```

```

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

```

```

284         }
285     }
286     else if (substitution.Count == 3)
287     {
288         Links.Update(after);
289     }
290     else
291     {
292         throw new NotSupportedException();
293     }
294     if (matchHandler != null)
295     {
296         return substitutionHandler(before, after);
297     }
298     return Constants.Continue;
299 }
300 else
301 {
302     throw new NotSupportedException();
303 }
304 }
305 }
306
307 /// <remarks>
308 /// IList<IList<IList<T>>>
309 /// |         |         |         |
310 /// |         |         |-----|
311 /// |         |         |   link   |
312 /// |         |-----|
313 /// |         |   change   |
314 /// |-----|
315 /// |   changes   |
316 /// </remarks>
317 public IList<IList<IList<TLink>>> Trigger(IList<TLink>
↪ condition, 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
8  ↪ (IRefEqualityComparer)
9  /// 2x faster with comparer
10 /// </remarks>
11 public class DoubletComparer<T> : IEqualityComparer<Doublet<T>>
12 {
13     private static readonly EqualityComparer<T> _equalityComparer
14     ↪ = EqualityComparer<T>.Default;
15
16     public static readonly DoubletComparer<T> Default = new
17     ↪ DoubletComparer<T>();
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     public bool Equals(Doublet<T> x, Doublet<T> y) =>
21     ↪ _equalityComparer.Equals(x.Source, y.Source) &&
22     ↪ _equalityComparer.Equals(x.Target, y.Target);
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     public int GetHashCode(Doublet<T> obj) =>
26     ↪ unchecked(obj.Source.GetHashCode() << 15 ^
27     ↪ obj.Target.GetHashCode());
28 }
29 }

```

./Doublet.cs

```

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

```

./Hybrid.cs

```

1  using System;
2  using System.Reflection;
3  using Platform.Reflection;
4  using Platform.Converters;
5  using Platform.Numbers;
6
7  namespace Platform.Data.Doublets

```

```

8  {
9      public class Hybrid<T>
10     {
11         public readonly T Value;
12         public bool IsNothing => Convert.ToInt64(To.Signed(Value)) ==
13         ↪ 0;
14         public bool IsInternal => Convert.ToInt64(To.Signed(Value)) >
15         ↪ 0;
16         public bool IsExternal => Convert.ToInt64(To.Signed(Value)) <
17         ↪ 0;
18         public long AbsoluteValue =>
19         ↪ Math.Abs(Convert.ToInt64(To.Signed(Value)));
20
21         public Hybrid(T value)
22         {
23             if (CachedTypeInfo<T>.IsSigned)
24             {
25                 throw new NotSupportedException();
26             }
27             Value = value;
28         }
29
30         public Hybrid(object value) => Value =
31         ↪ To.UnsignedAs<T>(Convert.ChangeType(value,
32         ↪ CachedTypeInfo<T>.SignedVersion));
33
34         public Hybrid(object value, bool isExternal)
35         {
36             var signedType = CachedTypeInfo<T>.SignedVersion;
37             var signedValue = Convert.ChangeType(value, signedType);
38             var abs = typeof(MathHelpers).GetTypeInfo().GetMethod("Abs"]
39             ↪ ").MakeGenericMethod(signedType);
40             var negate = typeof(MathHelpers).GetTypeInfo().GetMethod("]
41             ↪ Negate").MakeGenericMethod(signedType);
42             var absoluteValue = abs.Invoke(null, new[] { signedValue
43             ↪ });
44             var resultValue = isExternal ? negate.Invoke(null, new[] {
45             ↪ absoluteValue }) : absoluteValue;
46             Value = To.UnsignedAs<T>(resultValue);
47         }
48
49         public static implicit operator Hybrid<T>(T integer) => new
50         ↪ Hybrid<T>(integer);
51
52         public static explicit operator Hybrid<T>(ulong integer) =>
53         ↪ new Hybrid<T>(integer);
54
55         public static explicit operator Hybrid<T>(long integer) => new
56         ↪ Hybrid<T>(integer);
57
58         public static explicit operator Hybrid<T>(uint integer) => new
59         ↪ Hybrid<T>(integer);
60
61         public static explicit operator Hybrid<T>(int integer) => new
62         ↪ Hybrid<T>(integer);
63
64         public static explicit operator Hybrid<T>(ushort integer) =>
65         ↪ new Hybrid<T>(integer);
66     }
67 }

```

```

51     public static explicit operator Hybrid<T>(short integer) =>
        ↳ new Hybrid<T>(integer);
52
53     public static explicit operator Hybrid<T>(byte integer) => new
        ↳ Hybrid<T>(integer);
54
55     public static explicit operator Hybrid<T>(sbyte integer) =>
        ↳ new Hybrid<T>(integer);
56
57     public static implicit operator T(Hybrid<T> hybrid) =>
        ↳ hybrid.Value;
58
59     public static explicit operator ulong(Hybrid<T> hybrid) =>
        ↳ Convert.ToUInt64(hybrid.Value);
60
61     public static explicit operator long(Hybrid<T> hybrid) =>
        ↳ hybrid.AbsoluteValue;
62
63     public static explicit operator uint(Hybrid<T> hybrid) =>
        ↳ Convert.ToUInt32(hybrid.Value);
64
65     public static explicit operator int(Hybrid<T> hybrid) =>
        ↳ Convert.ToInt32(hybrid.AbsoluteValue);
66
67     public static explicit operator ushort(Hybrid<T> hybrid) =>
        ↳ Convert.ToUInt16(hybrid.Value);
68
69     public static explicit operator short(Hybrid<T> hybrid) =>
        ↳ Convert.ToInt16(hybrid.AbsoluteValue);
70
71     public static explicit operator byte(Hybrid<T> hybrid) =>
        ↳ Convert.ToByte(hybrid.Value);
72
73     public static explicit operator sbyte(Hybrid<T> hybrid) =>
        ↳ Convert.ToSByte(hybrid.AbsoluteValue);
74
75     public override string ToString() => IsNothing ? default(T) ==
        ↳ null ? "Nothing" : default(T).ToString() : IsExternal ?
        ↳ $"{<AbsoluteValue>}" : Value.ToString();
76 }
77 }

```

./ILinks.cs

```

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

```

./ILinksExtensions.cs

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Runtime.CompilerServices;
6 using Platform.Ranges;

```

```

7 using Platform.Collections.Arrays;
8 using Platform.Numbers;
9 using Platform.Random;
10 using Platform.Helpers.Setters;
11 using Platform.Data.Exceptions;
12
13 namespace Platform.Data.Doublets
14 {
15     public static class ILinksExtensions
16     {
17         public static void RunRandomCreations<TLink>(this
        ↳ ILinks<TLink> links, long amountOfCreations)
18         {
19             for (long i = 0; i < amountOfCreations; i++)
20             {
21                 var linksAddressRange = new Range<ulong>(0,
        ↳ (Integer<TLink>)links.Count());
22                 Integer<TLink> source = RandomHelpers.Default.NextUInt
        ↳ 64(linksAddressRange);
23                 Integer<TLink> target = RandomHelpers.Default.NextUInt
        ↳ 64(linksAddressRange);
24                 links.CreateAndUpdate(source, target);
25             }
26         }
27
28         public static void RunRandomSearches<TLink>(this ILinks<TLink>
        ↳ links, long amountOfSearches)
29         {
30             for (long i = 0; i < amountOfSearches; i++)
31             {
32                 var linkAddressRange = new Range<ulong>(1,
        ↳ (Integer<TLink>)links.Count());
33                 Integer<TLink> source =
        ↳ RandomHelpers.Default.NextUInt64(linkAddressRange);
34                 Integer<TLink> target =
        ↳ RandomHelpers.Default.NextUInt64(linkAddressRange);
35                 links.SearchOrDefault(source, target);
36             }
37         }
38
39         public static void RunRandomDeletions<TLink>(this
        ↳ ILinks<TLink> links, long amountOfDeletions)
40         {
41             var min = (ulong)amountOfDeletions >
        ↳ (Integer<TLink>)links.Count() ? 1 :
        ↳ (Integer<TLink>)links.Count() -
        ↳ (ulong)amountOfDeletions;
42             for (long i = 0; i < amountOfDeletions; i++)
43             {
44                 var linksAddressRange = new Range<ulong>(min,
        ↳ (Integer<TLink>)links.Count());
45                 Integer<TLink> link = RandomHelpers.Default.NextUInt64
        ↳ (linksAddressRange);
46                 links.Delete(link);
47                 if ((Integer<TLink>)links.Count() < min)
48                 {
49                     break;
50                 }

```

```

51     }
52 }
53
54 /// <remarks>
55 /// TODO: Возможно есть очень простой способ это сделать.
56 /// (Например просто удалить файл, или изменить его размер
57   ↳ таким образом,
58   ↳ чтобы удалился весь контент)
59 /// Например через _header->AllocatedLinks в
60   ↳ ResizableDirectMemoryLinks
61 /// </remarks>
62 public static void DeleteAll<TLink>(this ILinks<TLink> links)
63 {
64     var equalityComparer = EqualityComparer<TLink>.Default;
65     var comparer = Comparer<TLink>.Default;
66     for (var i = links.Count(); comparer.Compare(i, default) >
67         ↳ 0; i = ArithmeticHelpers.Decrement(i))
68     {
69         links.Delete(i);
70         if (!equalityComparer.Equals(links.Count(),
71             ↳ ArithmeticHelpers.Decrement(i)))
72         {
73             i = links.Count();
74         }
75     }
76 }
77
78 public static TLink First<TLink>(this ILinks<TLink> links)
79 {
80     TLink firstLink = default;
81     var equalityComparer = EqualityComparer<TLink>.Default;
82     if (equalityComparer.Equals(links.Count(), default))
83     {
84         throw new Exception("В хранилище нет связей.");
85     }
86     links.Each(links.Constants.Any, links.Constants.Any, link
87         ↳ =>
88     {
89         firstLink = link[links.Constants.IndexPart];
90         return links.Constants.Break;
91     });
92     if (equalityComparer.Equals(firstLink, default))
93     {
94         throw new Exception("В процессе поиска по хранилищу не
95             ↳ было найдено связей.");
96     }
97     return firstLink;
98 }
99
100 public static bool IsInnerReference<TLink>(this ILinks<TLink>
101     ↳ links, TLink reference)
102 {
103     var constants = links.Constants;
104     var comparer = Comparer<TLink>.Default;
105     return comparer.Compare(constants.MinPossibleIndex,
106         ↳ reference) >= 0 && comparer.Compare(reference,
107         ↳ constants.MaxPossibleIndex) <= 0;
108 }
109
110 #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
107   ↳ в проходе через SequenceWalker
108   ↳ Stack в конкретный путь из Source, Target до связи, но это
109   ↳ не всегда так.
110 /// TODO: Возможно нужен метод, который именно выбрасывает
111   ↳ исключения (EnsurePathExists)
112 /// </remarks>
113 public static bool CheckPathExistance<TLink>(this
114     ↳ ILinks<TLink> links, params TLink[] path)
115 {
116     var current = path[0];
117     //EnsureLinkExists(current, "path");
118     if (!links.Exists(current))
119     {
120         return false;
121     }
122     var equalityComparer = EqualityComparer<TLink>.Default;
123     var constants = links.Constants;
124     for (var i = 1; i < path.Length; i++)
125     {
126         var next = path[i];
127         var values = links.GetLink(current);
128         var source = values[constants.SourcePart];
129         var target = values[constants.TargetPart];
130         if (equalityComparer.Equals(source, target) &&
131             ↳ equalityComparer.Equals(source, next))
132         {
133             //throw new Exception(string.Format("Невозможно
134                 ↳ выбрать путь, так как и Source и Target
135                 ↳ совпадают с элементом пути {0}.", next));
136             return false;
137         }
138         if (!equalityComparer.Equals(next, source) &&
139             ↳ !equalityComparer.Equals(next, target))
140         {
141             //throw new Exception(string.Format("Невозможно
142                 ↳ продолжить путь через элемент пути {0}",
143                 ↳ next));
144             return false;
145         }
146         current = next;
147     }
148     return true;
149 }
150
151 /// <remarks>
152 /// Может потребовать дополнительного стека для PathElement's
153   ↳ при использовании SequenceWalker.
154 /// </remarks>
155 public static TLink GetByKeyes<TLink>(this ILinks<TLink> links,
156     ↳ TLink root, params int[] path)
157 {
158     links.EnsureLinkExists(root, "root");
159     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
156     ↳ GetSquareMatrixSequenceElementByIndex<TLink>(this
157     ↳ ILinks<TLink> links, TLink root, ulong size, ulong index)
158 {
159     var constants = links.Constants;
160     var source = constants.SourcePart;
161     var target = constants.TargetPart;
162     if (!MathHelpers.IsPowerOfTwo(size))
163     {
164         throw new ArgumentOutOfRangeException(nameof(size),
165             ↳ "Sequences with sizes other than powers of two are
166             ↳ not supported.");
167     }
168     var path = new BitArray(BitConverter.GetBytes(index));
169     var length = BitwiseHelpers.GetLowestBitPosition(size);
170     links.EnsureLinkExists(root, "root");
171     var currentLink = root;
172     for (var i = length - 1; i >= 0; i--)
173     {
174         currentLink = links.GetLink(currentLink)[path[i] ?
175             ↳ target : source];
176     }
177     return currentLink;
178 }
179
180 #endregion
181
182 /// <summary>
183 /// Возвращает индекс указанной связи.
184 /// </summary>
185 /// <param name="links">Хранилище связей.</param>
186 /// <param name="link">Связь представленная списком, состоящим
187   ↳ из её адреса и содержимого.</param>
188 /// <returns>Индекс начальной связи для указанной
189   ↳ связи.</returns>
190 [MethodImpl(MethodImplOptions.AggressiveInlining)]
191 public static TLink GetIndex<TLink>(this ILinks<TLink> links,
192     ↳ IList<TLink> link) => link[links.Constants.IndexPart];
193
194 /// <summary>
195 /// Возвращает индекс начальной (Source) связи для указанной
196   ↳ связи.
197 /// </summary>
198 /// <param name="links">Хранилище связей.</param>
199 /// <param name="link">Индекс связи.</param>
200 /// <returns>Индекс начальной связи для указанной
201   ↳ связи.</returns>
202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 public static TLink GetSource<TLink>(this ILinks<TLink> links,
204     ↳ TLink link) =>
205     links.GetLink(link)[links.Constants.SourcePart];
206
207 /// <summary>
208 /// Возвращает индекс конечной (Target) связи для указанной
209   ↳ связи.
210 /// </summary>
211 /// <param name="links">Хранилище связей.</param>
212 /// <param name="link">Индекс связи.</param>
213 /// <returns>Индекс конечной связи для указанной
214   ↳ связи.</returns>
215 [MethodImpl(MethodImplOptions.AggressiveInlining)]
216 public static TLink GetTarget<TLink>(this ILinks<TLink> links,
217     ↳ TLink link) =>
218     links.GetLink(link)[links.Constants.TargetPart];
219
220 /// <summary>
221 /// Возвращает индекс конечной (Target) связи для указанной
222   ↳ связи.
223 /// </summary>
224 /// <param name="links">Хранилище связей.</param>
225 /// <param name="link">Связь представленная списком, состоящим
226   ↳ из её адреса и содержимого.</param>
227 /// <returns>Индекс конечной связи для указанной
228   ↳ связи.</returns>
229 [MethodImpl(MethodImplOptions.AggressiveInlining)]
230 public static TLink GetTarget<TLink>(this ILinks<TLink> links,
231     ↳ IList<TLink> link) => link[links.Constants.TargetPart];
232
233 /// <summary>
234 /// Выполняет проход по всем связям, соответствующим шаблону,
235   ↳ вызывая обработчик (handler) для каждой подходящей связи.
236 /// </summary>
237 /// <param name="links">Хранилище связей.</param>
238 /// <param name="handler">Обработчик каждой подходящей
239   ↳ связи.</param>
240 /// <param name="restrictions">Ограничения на содержимое
241   ↳ связей. Каждое ограничение может иметь значения:
242   ↳ Constants.Null - 0-я связь, обозначающая ссылку на
243   ↳ пустоту, Any - отсутствие ограничения, 1..∞ конкретный
244   ↳ адрес связи.</param>
245 /// <returns>True, в случае если проход по связям не был
246   ↳ прерван и False в обратном случае.</returns>
247 [MethodImpl(MethodImplOptions.AggressiveInlining)]
248 public static bool Each<TLink>(this ILinks<TLink> links,
249     ↳ Func<IList<TLink>, TLink> handler, params TLink[]
250     ↳ restrictions)

```

```

249     /// Возвращает индекс начальной (Source) связи для указанной
250     ↳ связи.
251     /// </summary>
252     /// <param name="links">Хранилище связей.</param>
253     /// <param name="link">Связь представленная списком, состоящим
254     ↳ из её адреса и содержимого.</param>
255     /// <returns>Индекс начальной связи для указанной
256     ↳ связи.</returns>
257     [MethodImpl(MethodImplOptions.AggressiveInlining)]
258     public static TLink GetSource<TLink>(this ILinks<TLink> links,
259     ↳ IList<TLink> link) => link[links.Constants.SourcePart];
260
261     /// <summary>
262     /// Возвращает индекс конечной (Target) связи для указанной
263     ↳ связи.
264     /// </summary>
265     /// <param name="links">Хранилище связей.</param>
266     /// <param name="link">Индекс связи.</param>
267     /// <returns>Индекс конечной связи для указанной
268     ↳ связи.</returns>
269     [MethodImpl(MethodImplOptions.AggressiveInlining)]
270     public static TLink GetTarget<TLink>(this ILinks<TLink> links,
271     ↳ TLink link) =>
272     ↳ links.GetLink(link)[links.Constants.TargetPart];
273
274     /// <summary>
275     /// Возвращает индекс конечной (Target) связи для указанной
276     ↳ связи.
277     /// </summary>
278     /// <param name="links">Хранилище связей.</param>
279     /// <param name="link">Связь представленная списком, состоящим
280     ↳ из её адреса и содержимого.</param>
281     /// <returns>Индекс конечной связи для указанной
282     ↳ связи.</returns>
283     [MethodImpl(MethodImplOptions.AggressiveInlining)]
284     public static TLink GetTarget<TLink>(this ILinks<TLink> links,
285     ↳ IList<TLink> link) => link[links.Constants.TargetPart];
286
287     /// <summary>
288     /// Выполняет проход по всем связям, соответствующим шаблону,
289     ↳ вызывая обработчик (handler) для каждой подходящей связи.
290     /// </summary>
291     /// <param name="links">Хранилище связей.</param>
292     /// <param name="handler">Обработчик каждой подходящей
293     ↳ связи.</param>
294     /// <param name="restrictions">Ограничения на содержимое
295     ↳ связей. Каждое ограничение может иметь значения:
296     ↳ Constants.Null - 0-я связь, обозначающая ссылку на
297     ↳ пустоту, Any - отсутствие ограничения, 1..∞ конкретный
298     ↳ адрес связи.</param>
299     /// <returns>True, в случае если проход по связям не был
300     ↳ прерван и False в обратном случае.</returns>
301     [MethodImpl(MethodImplOptions.AggressiveInlining)]
302     public static bool Each<TLink>(this ILinks<TLink> links,
303     ↳ Func<IList<TLink>, TLink> handler, params TLink[]
304     ↳ restrictions)

```

```

231     => EqualityComparer<TLink>.Default.Equals(links.Each(handler,
232         ↪ er, restrictions),
233         ↪ links.Constants.Continue);
234
235     /// <summary>
236     /// Выполняет проход по всем связям, соответствующим шаблону,
237     ↪ вызывая обработчик (handler) для каждой подходящей связи.
238     /// </summary>
239     /// <param name="links">Хранилище связей.</param>
240     /// <param name="source">Значение, определяющее
241     ↪ соответствующие шаблону связи. (Constants.Null - 0-я
242     ↪ связь, обозначающая ссылку на пустоту в качестве начала,
243     ↪ Constants.Any - любое начало, 1..∞ конкретное
244     ↪ начало)</param>
245     /// <param name="target">Значение, определяющее
246     ↪ соответствующие шаблону связи. (Constants.Null - 0-я
247     ↪ связь, обозначающая ссылку на пустоту в качестве конца,
248     ↪ Constants.Any - любой конец, 1..∞ конкретный
249     ↪ конец)</param>
250     /// <param name="handler">Обработчик каждой подходящей
251     ↪ связи.</param>
252     /// <returns>True, в случае если проход по связям не был
253     ↪ прерван и False в обратном случае.</returns>
254     [MethodImpl(MethodImplOptions.AggressiveInlining)]
255     public static bool Each<TLink>(this ILinks<TLink> links, TLink
256     ↪ source, TLink target, Func<TLink, bool> handler)
257     {
258         var constants = links.Constants;
259         return links.Each(link =>
260             ↪ handler(link[constants.IndexPart]) ?
261             ↪ constants.Continue : constants.Break, constants.Any,
262             ↪ source, target);
263     }
264
265     /// <summary>
266     /// Выполняет проход по всем связям, соответствующим шаблону,
267     ↪ вызывая обработчик (handler) для каждой подходящей связи.
268     /// </summary>
269     /// <param name="links">Хранилище связей.</param>
270     /// <param name="source">Значение, определяющее
271     ↪ соответствующие шаблону связи. (Constants.Null - 0-я
272     ↪ связь, обозначающая ссылку на пустоту в качестве начала,
273     ↪ Constants.Any - любое начало, 1..∞ конкретное
274     ↪ начало)</param>
275     /// <param name="target">Значение, определяющее
276     ↪ соответствующие шаблону связи. (Constants.Null - 0-я
277     ↪ связь, обозначающая ссылку на пустоту в качестве конца,
278     ↪ Constants.Any - любой конец, 1..∞ конкретный
279     ↪ конец)</param>
280     /// <param name="handler">Обработчик каждой подходящей
281     ↪ связи.</param>
282     /// <returns>True, в случае если проход по связям не был
283     ↪ прерван и False в обратном случае.</returns>
284     [MethodImpl(MethodImplOptions.AggressiveInlining)]
285     public static bool Each<TLink>(this ILinks<TLink> links, TLink
286     ↪ source, TLink target, Func<IList<TLink>, TLink> handler)
287     {
288         var constants = links.Constants;
289         return links.Each(link =>
290             ↪ handler(link[constants.IndexPart]) ?
291             ↪ constants.Continue : constants.Break, constants.Any,
292             ↪ source, target);
293     }
294
295     /// <summary>
296     /// Возвращает значение, определяющее существует ли связь с
297     ↪ указанными началом и концом в хранилище связей.
298     /// </summary>
299     /// <param name="links">Хранилище связей.</param>
300     /// <param name="source">Начало связи.</param>
301     /// <param name="target">Конец связи.</param>
302     /// <returns>Значение, определяющее существует ли
303     ↪ связь.</returns>
304     [MethodImpl(MethodImplOptions.AggressiveInlining)]
305     public static bool Exists<TLink>(this ILinks<TLink> links,
306     ↪ TLink source, TLink target) => Comparer<TLink>.Default.Com
307     ↪ pare(links.Count(links.Constants.Any, source, target),
308     ↪ default) > 0;
309
310     #region Ensure
311     // TODO: May be move to EnsureExtensions or make it both there
312     ↪ and here
313
314     [MethodImpl(MethodImplOptions.AggressiveInlining)]
315     public static void EnsureInnerReferenceExists<TLink>(this
316     ↪ ILinks<TLink> links, TLink reference, string argumentName)
317     {
318         if (links.IsInnerReference(reference) &&
319             ↪ !links.Exists(reference))
320         {
321             throw new ArgumentLinkDoesNotExistsException<TLink>(re
322             ↪ ference,
323             ↪ argumentName);
324         }
325     }
326
327     [MethodImpl(MethodImplOptions.AggressiveInlining)]
328     public static void EnsureInnerReferenceExists<TLink>(this
329     ↪ ILinks<TLink> links, IList<TLink> restrictions, string
330     ↪ argumentName)
331     {
332         for (int i = 0; i < restrictions.Count; i++)
333         {

```

```

334             return links.Each(handler, constants.Any, source, target);
335         }
336     }
337
338     [MethodImpl(MethodImplOptions.AggressiveInlining)]
339     public static IList<IList<TLink>> All<TLink>(this
340     ↪ ILinks<TLink> links, params TLink[] restrictions)
341     {
342         var constants = links.Constants;
343         int listSize = (Integer<TLink>)links.Count(restrictions);
344         var list = new IList<TLink>[listSize];
345         if (listSize > 0)
346         {
347             var filler = new ArrayFiller<IList<TLink>,
348             ↪ TLink>(list, links.Constants.Continue);
349             links.Each(filler.AddAndReturnConstant, restrictions);
350         }
351         return list;
352     }
353
354     /// <summary>
355     /// Возвращает значение, определяющее существует ли связь с
356     ↪ указанными началом и концом в хранилище связей.
357     /// </summary>
358     /// <param name="links">Хранилище связей.</param>
359     /// <param name="source">Начало связи.</param>
360     /// <param name="target">Конец связи.</param>
361     /// <returns>Значение, определяющее существует ли
362     ↪ связь.</returns>
363     [MethodImpl(MethodImplOptions.AggressiveInlining)]
364     public static bool Exists<TLink>(this ILinks<TLink> links,
365     ↪ TLink source, TLink target) => Comparer<TLink>.Default.Com
366     ↪ pare(links.Count(links.Constants.Any, source, target),
367     ↪ default) > 0;
368
369     #region Ensure
370     // TODO: May be move to EnsureExtensions or make it both there
371     ↪ and here
372
373     [MethodImpl(MethodImplOptions.AggressiveInlining)]
374     public static void EnsureInnerReferenceExists<TLink>(this
375     ↪ ILinks<TLink> links, TLink reference, string argumentName)
376     {
377         if (links.IsInnerReference(reference) &&
378             ↪ !links.Exists(reference))
379         {
380             throw new ArgumentLinkDoesNotExistsException<TLink>(re
381             ↪ ference,
382             ↪ argumentName);
383         }
384     }
385
386     [MethodImpl(MethodImplOptions.AggressiveInlining)]
387     public static void EnsureInnerReferenceExists<TLink>(this
388     ↪ ILinks<TLink> links, IList<TLink> restrictions, string
389     ↪ argumentName)
390     {
391         for (int i = 0; i < restrictions.Count; i++)
392         {

```



```

304         links.EnsureInnerReferenceExists(restrictions[i],
305             ↳ argumentName);
306     }
307 }
308 [MethodImpl(MethodImplOptions.AggressiveInlining)]
309 public static void EnsureLinkIsAnyOrExists<TLink>(this
310     ↳ ILinks<TLink> links, IList<TLink> restrictions)
311 {
312     for (int i = 0; i < restrictions.Count; i++)
313     {
314         links.EnsureLinkIsAnyOrExists(restrictions[i],
315             ↳ nameof(restrictions));
316     }
317 }
318 [MethodImpl(MethodImplOptions.AggressiveInlining)]
319 public static void EnsureLinkIsAnyOrExists<TLink>(this
320     ↳ ILinks<TLink> links, TLink link, string argumentName)
321 {
322     var equalityComparer = EqualityComparer<TLink>.Default;
323     if (!equalityComparer.Equals(link, links.Constants.Any) &&
324         ↳ !links.Exists(link))
325     {
326         throw new
327             ↳ ArgumentLinkDoesNotExistsException<TLink>(link,
328                 ↳ argumentName);
329     }
330 }
331 [MethodImpl(MethodImplOptions.AggressiveInlining)]
332 public static void EnsureLinkIsItselfOrExists<TLink>(this
333     ↳ ILinks<TLink> links, TLink link, string argumentName)
334 {
335     var equalityComparer = EqualityComparer<TLink>.Default;
336     if (!equalityComparer.Equals(link, links.Constants.Itself)
337         ↳ && !links.Exists(link))
338     {
339         throw new
340             ↳ ArgumentLinkDoesNotExistsException<TLink>(link,
341                 ↳ argumentName);
342     }
343 }
344 }
345 }
346
347 /// <param name="links">Хранилище связей.</param>
348 [MethodImpl(MethodImplOptions.AggressiveInlining)]
349 public static void EnsureDoesNotExists<TLink>(this
350     ↳ ILinks<TLink> links, TLink source, TLink target)
351 {
352     if (links.Exists(source, target))
353     {
354         throw new LinkWithSameValueAlreadyExistsException();
355     }
356 }
357
358 /// <param name="links">Хранилище связей.</param>
359 public static void EnsureNoDependencies<TLink>(this
360     ↳ ILinks<TLink> links, TLink link)
361 {

```

```

350     if (links.DependenciesExist(link))
351     {
352         throw new
353             ↳ ArgumentLinkHasDependenciesException<TLink>(link);
354     }
355 }
356
357 /// <param name="links">Хранилище связей.</param>
358 public static void EnsureCreated<TLink>(this ILinks<TLink>
359     ↳ links, params TLink[] addresses) =>
360     ↳ links.EnsureCreated(links.Create, addresses);
361
362 /// <param name="links">Хранилище связей.</param>
363 public static void EnsurePointsCreated<TLink>(this
364     ↳ ILinks<TLink> links, params TLink[] addresses) =>
365     ↳ links.EnsureCreated(links.CreatePoint, addresses);
366
367 /// <param name="links">Хранилище связей.</param>
368 public static void EnsureCreated<TLink>(this ILinks<TLink>
369     ↳ links, Func<TLink> creator, params TLink[] addresses)
370 {
371     var constants = links.Constants;
372     var nonExistentAddresses = new
373         ↳ HashSet<ulong>(addresses.Where(x =>
374             ↳ !links.Exists(x)).Select(x =>
375                 ↳ (ulong)(Integer<TLink>)x));
376     if (nonExistentAddresses.Count > 0)
377     {
378         var max = nonExistentAddresses.Max();
379         // TODO: Эту верхнюю границу нужно разрешить
380         ↳ переопределять (проверить применяется ли эта
381         ↳ логика)
382         max = Math.Min(max,
383             ↳ (Integer<TLink>)constants.MaxPossibleIndex);
384         var createdLinks = new List<TLink>();
385         var equalityComparer = EqualityComparer<TLink>.Default;
386         TLink createdLink = creator();
387         while (!equalityComparer.Equals(createdLink,
388             ↳ (Integer<TLink>)max))
389         {
390             createdLinks.Add(createdLink);
391         }
392         for (var i = 0; i < createdLinks.Count; i++)
393         {
394             if (!nonExistentAddresses.Contains((Integer<TLink>
395                 ↳ )createdLinks[i]))
396             {
397                 links.Delete(createdLinks[i]);
398             }
399         }
400     }
401 }
402
403 #endregion
404
405 /// <param name="links">Хранилище связей.</param>
406 public static ulong DependenciesCount<TLink>(this
407     ↳ ILinks<TLink> links, TLink link)

```

```

393 {
394     var constants = links.Constants;
395     var values = links.GetLink(link);
396     ulong referencesAsSource =
397         ↪ (Integer<TLink>)links.Count(constants.Any, link,
398         ↪ constants.Any);
399     var equalityComparer = EqualityComparer<TLink>.Default;
400     if (equalityComparer.Equals(values[constants.SourcePart],
401     ↪ link))
402     {
403         referencesAsSource--;
404     }
405     ulong referencesAsTarget =
406         ↪ (Integer<TLink>)links.Count(constants.Any,
407         ↪ constants.Any, link);
408     if (equalityComparer.Equals(values[constants.TargetPart],
409     ↪ link))
410     {
411         referencesAsTarget--;
412     }
413     return referencesAsSource + referencesAsTarget;
414 }
415
416 /// <param name="links">Хранилище связей.</param>
417 [MethodImpl(MethodImplOptions.AggressiveInlining)]
418 public static bool DependenciesExist<TLink>(this ILinks<TLink>
419 ↪ links, TLink link) => links.DependenciesCount(link) > 0;
420
421 /// <param name="links">Хранилище связей.</param>
422 [MethodImpl(MethodImplOptions.AggressiveInlining)]
423 public static bool Equals<TLink>(this ILinks<TLink> links,
424 ↪ TLink link, TLink source, TLink target)
425 {
426     var constants = links.Constants;
427     var values = links.GetLink(link);
428     var equalityComparer = EqualityComparer<TLink>.Default;
429     return
430         ↪ equalityComparer.Equals(values[constants.SourcePart],
431         ↪ source) &&
432         ↪ equalityComparer.Equals(values[constants.TargetPart],
433         ↪ target);
434 }
435
436 /// <summary>
437 /// Выполняет поиск связи с указанными Source (началом) и
438 ↪ Target (концом).
439 /// </summary>
440 /// <param name="links">Хранилище связей.</param>
441 /// <param name="source">Индекс связи, которая является
442 ↪ началом для искомой связи.</param>
443 /// <param name="target">Индекс связи, которая является концом
444 ↪ для искомой связи.</param>
445 /// <returns>Индекс искомой связи с указанными Source
446 ↪ (началом) и Target (концом).</returns>
447 [MethodImpl(MethodImplOptions.AggressiveInlining)]
448 public static TLink SearchOrDefault<TLink>(this ILinks<TLink>
449 ↪ links, TLink source, TLink target)
450 {
451     var constants = links.Constants;

```

```

452     var setter = new Setter<TLink, TLink>(constants.Continue,
453     ↪ constants.Break, default);
454     links.Each(setter.SetFirstAndReturnFalse, constants.Any,
455     ↪ source, target);
456     return setter.Result;
457 }
458
459 /// <param name="links">Хранилище связей.</param>
460 [MethodImpl(MethodImplOptions.AggressiveInlining)]
461 public static TLink CreatePoint<TLink>(this ILinks<TLink>
462 ↪ links)
463 {
464     var link = links.Create();
465     return links.Update(link, link, link);
466 }
467
468 /// <param name="links">Хранилище связей.</param>
469 [MethodImpl(MethodImplOptions.AggressiveInlining)]
470 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink>
471 ↪ links, TLink source, TLink target) =>
472     ↪ links.Update(links.Create(), source, target);
473
474 /// <summary>
475 /// Обновляет связь с указанными началом (Source) и концом
476 ↪ (Target)
477 /// на связь с указанными началом (NewSource) и концом
478 ↪ (NewTarget).
479 /// </summary>
480 /// <param name="links">Хранилище связей.</param>
481 /// <param name="link">Индекс обновляемой связи.</param>
482 /// <param name="newSource">Индекс связи, которая является
483 ↪ началом связи, на которую выполняется обновление.</param>
484 /// <param name="newTarget">Индекс связи, которая является
485 ↪ концом связи, на которую выполняется обновление.</param>
486 /// <returns>Индекс обновлённой связи.</returns>
487 [MethodImpl(MethodImplOptions.AggressiveInlining)]
488 public static TLink Update<TLink>(this ILinks<TLink> links,
489 ↪ TLink link, TLink newSource, TLink newTarget) =>
490     ↪ links.Update(new[] { link, newSource, newTarget });
491
492 /// <summary>
493 /// Обновляет связь с указанными началом (Source) и концом
494 ↪ (Target)
495 /// на связь с указанными началом (NewSource) и концом
496 ↪ (NewTarget).
497 /// </summary>
498 /// <param name="links">Хранилище связей.</param>
499 /// <param name="restrictions">Ограничения на содержимое
500 ↪ связей. Каждое ограничение может иметь значения:
501 ↪ Constants.Null - 0-я связь, обозначающая ссылку на
502 ↪ пустоту, Itself - требование установить ссылку на себя,
503 ↪ 1..∞ конкретный адрес другой связи.</param>
504 /// <returns>Индекс обновлённой связи.</returns>
505 [MethodImpl(MethodImplOptions.AggressiveInlining)]
506 public static TLink Update<TLink>(this ILinks<TLink> links,
507 ↪ params TLink[] restrictions)
508 {
509     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],
481             ↪ restrictions[1], restrictions[2], restrictions[3]);
482     }
483     else
484     {
485         return links.Update(restrictions);
486     }
487 }
488
489 /// <summary>
490 /// Создаёт связь (если она не существовала), либо возвращает
491 ↪ индекс существующей связи с указанными Source (началом) и
492 ↪ Target (концом).
493 /// </summary>
494 /// <param name="links">Хранилище связей.</param>
495 /// <param name="source">Индекс связи, которая является
496 ↪ началом на создаваемой связи.</param>
497 /// <param name="target">Индекс связи, которая является концом
498 ↪ для создаваемой связи.</param>
499 /// <returns>Индекс связи, с указанным Source (началом) и
500 ↪ Target (концом)</returns>
501 [MethodImpl(MethodImplOptions.AggressiveInlining)]
502 public static TLink GetOrCreate<TLink>(this ILinks<TLink>
503 ↪ links, TLink source, TLink target)
504 {
505     var link = links.SearchOrDefault(source, target);
506     if (EqualityComparer<TLink>.Default.Equals(link, default))
507     {
508         link = links.CreateAndUpdate(source, target);
509     }
510     return link;
511 }
512
513 /// <summary>
514 /// Обновляет связь с указанными началом (Source) и концом
515 ↪ (Target)
516 /// на связь с указанными началом (NewSource) и концом
517 ↪ (NewTarget).
518 /// </summary>
519 /// <param name="links">Хранилище связей.</param>
520 /// <param name="source">Индекс связи, которая является
521 ↪ началом обновляемой связи.</param>
522 /// <param name="target">Индекс связи, которая является концом
523 ↪ обновляемой связи.</param>
524 /// <param name="newSource">Индекс связи, которая является
525 ↪ началом связи, на которую выполняется обновление.</param>
526 /// <param name="newTarget">Индекс связи, которая является
527 ↪ концом связи, на которую выполняется обновление.</param>
528 /// <returns>Индекс обновлённой связи.</returns>
529 [MethodImpl(MethodImplOptions.AggressiveInlining)]
530 public static TLink UpdateOrCreateOrGet<TLink>(this
531 ↪ ILinks<TLink> links, TLink source, TLink target, TLink
532 ↪ 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) &&
526             ↪ equalityComparer.Equals(newTarget, target))
527         {
528             return link;
529         }
530         return links.Update(link, newSource, newTarget);
531     }
532 }
533
534 /// <summary>Удаляет связь с указанными началом (Source) и
535 ↪ концом (Target).</summary>
536 /// <param name="links">Хранилище связей.</param>
537 /// <param name="source">Индекс связи, которая является
538 ↪ началом удаляемой связи.</param>
539 /// <param name="target">Индекс связи, которая является концом
540 ↪ удаляемой связи.</param>
541 [MethodImpl(MethodImplOptions.AggressiveInlining)]
542 public static TLink DeleteIfExists<TLink>(this ILinks<TLink>
543 ↪ links, TLink source, TLink target)
544 {
545     var link = links.SearchOrDefault(source, target);
546     if (!EqualityComparer<TLink>.Default.Equals(link, default))
547     {
548         links.Delete(link);
549         return link;
550     }
551     return default;
552 }
553
554 /// <summary>Удаляет несколько связей.</summary>
555 /// <param name="links">Хранилище связей.</param>
556 /// <param name="deletedLinks">Список адресов связей к
557 ↪ удалению.</param>
558 [MethodImpl(MethodImplOptions.AggressiveInlining)]
559 public static void DeleteMany<TLink>(this ILinks<TLink> links,
560 ↪ IList<TLink> deletedLinks)
561 {
562     for (int i = 0; i < deletedLinks.Count; i++)
563     {
564         links.Delete(deletedLinks[i]);
565     }
566 }
567
568 // Replace one link with another (replaced link is deleted,
569 ↪ children are updated or deleted)
570 public static TLink Merge<TLink>(this ILinks<TLink> links,
571 ↪ TLink linkIndex, TLink newLink)
572 {
573     var equalityComparer = EqualityComparer<TLink>.Default;
574     if (equalityComparer.Equals(linkIndex, newLink))
575     {
576         return newLink;
577     }
578 }

```

```

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)t
    ↳ otalReferences);
578         var referencesFiller = new ArrayFiller<TLink,
    ↳ TLink>(references, links.Constants.Continue);
579         links.Each(referencesFiller.AddFirstAndReturnConst
    ↳ ant, constants.Any, linkIndex,
    ↳ constants.Any);
580         links.Each(referencesFiller.AddFirstAndReturnConst
    ↳ ant, 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,
    ↳ linkIndex, links.GetTarget(reference));
589         }
590         for (var i = (long)referencesAsSourceCount; i <
    ↳ references.Length; i++)
591         {
592             var reference = references[i];
593             if (equalityComparer.Equals(reference,
    ↳ linkIndex))
594             {
595                 continue;
596             }
597             links.Update(reference,
    ↳ linkIndex, links.GetSource(reference), newLink);
598         }
599         ArrayPool.Free(references);
600     }
601 }
602 links.Delete(linkIndex);
603 return newLink;
604 }
605 }
606

```

```

607 }
608 }

./Incrementers/FrequencyIncrementer.cs
1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Incrementers
5 {
6     public class FrequencyIncrementer<TLink> :
    ↳ LinksOperatorBase<TLink>, IIncrementer<TLink>
7     {
8         private static readonly EqualityComparer<TLink>
    ↳ _equalityComparer = EqualityComparer<TLink>.Default;
9
10        private readonly TLink _frequencyMarker;
11        private readonly TLink _unaryOne;
12        private readonly IIncrementer<TLink> _unaryNumberIncrementer;
13
14        public FrequencyIncrementer(ILinks<TLink> links, TLink
    ↳ frequencyMarker, TLink unaryOne, IIncrementer<TLink>
    ↳ unaryNumberIncrementer)
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>, IIncrementer<IList<TLink>>
7     {
8         private readonly ISpecificPropertyOperator<TLink, TLink>
    ↳ _frequencyPropertyOperator;
9         private readonly IIncrementer<TLink> _frequencyIncrementer;
10
11        public LinkFrequencyIncrementer(ILinks<TLink> links,
    ↳ ISpecificPropertyOperator<TLink, TLink>
    ↳ frequencyPropertyOperator, IIncrementer<TLink>
    ↳ frequencyIncrementer)

```

```

12         : base(links)
13     {
14         _frequencyPropertyOperator = frequencyPropertyOperator;
15         _frequencyIncrementer = frequencyIncrementer;
16     }
17
18     /// <remarks>Sequence itself is not changed, only frequency of
19     ↪ its doublets is incremented.</remarks>
20     public IList<TLink> Increment(IList<TLink> sequence) // TODO:
21     ↪ May be move to ILinksExtensions or make
22     ↪ SequenceDoubletsFrequencyIncrementer
23     {
24         for (var i = 1; i < sequence.Count; i++)
25         {
26             Increment(Links.GetOrCreate(sequence[i - 1],
27                 ↪ sequence[i]));
28         }
29         return sequence;
30     }
31
32     public void Increment(TLink link)
33     {
34         var previousFrequency =
35         ↪ _frequencyPropertyOperator.Get(link);
36         var frequency =
37         ↪ _frequencyIncrementer.Increment(previousFrequency);
38         _frequencyPropertyOperator.Set(link, frequency);
39     }
40 }

```

./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> :
7     ↪ LinksOperatorBase<TLink>, IIncrementer<TLink>
8     {
9         private static readonly EqualityComparer<TLink>
10         ↪ _equalityComparer = EqualityComparer<TLink>.Default;
11
12         private readonly TLink _unaryOne;
13
14         public UnaryNumberIncrementer(ILinks<TLink> links, TLink
15         ↪ unaryOne) : base(links) => _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 }

```

```

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> :
6     ↪ ISynchronizedLinks<TLink, ILinks<TLink>,
7     ↪ LinksCombinedConstants<TLink, TLink, int>>, ILinks<TLink>
8     {
9     }
10 }

```

./Link.cs

```

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

```

```

34 public Link(IList<TLink> values)
35 {
36     Index = values.Count > _constants.IndexPart ?
        ↳ values[_constants.IndexPart] : _constants.Null;
37     Source = values.Count > _constants.SourcePart ?
        ↳ values[_constants.SourcePart] : _constants.Null;
38     Target = values.Count > _constants.TargetPart ?
        ↳ values[_constants.TargetPart] : _constants.Null;
39 }
40
41 public Link(TLink index, TLink source, TLink target)
42 {
43     Index = index;
44     Source = source;
45     Target = target;
46 }
47
48 public Link(TLink source, TLink target)
49     : this(_constants.Null, source, target)
50 {
51     Source = source;
52     Target = target;
53 }
54
55 public static Link<TLink> Create(TLink source, TLink target)
56     ↳ => new Link<TLink>(source, target);
57
58 public override int GetHashCode() => (Index, Source,
59     ↳ Target).GetHashCode();
60
61 public bool IsNull() => _equalityComparer.Equals(Index,
62     ↳ _constants.Null)
63     && _equalityComparer.Equals(Source,
64         ↳ _constants.Null)
65     && _equalityComparer.Equals(Target,
66         ↳ _constants.Null);
67
68 public override bool Equals(object other) => other is
69     ↳ Link<TLink> && Equals((Link<TLink>)other);
70
71 public bool Equals(Link<TLink> other) =>
72     ↳ _equalityComparer.Equals(Index, other.Index)
73     && _equalityComparer.Equals(
74         Source, other.Source)
75     && _equalityComparer.Equals(
76         Target, other.Target);
77
78 public static string ToString(TLink index, TLink source, TLink
79     ↳ target) => $"{index}: {source}->{target}";
80
81 public static string ToString(TLink source, TLink target) =>
82     ↳ $"{source}->{target}";
83
84 public static implicit operator TLink[] (Link<TLink> link) =>
85     ↳ link.ToArray();

```

```

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

```

```

130     {
131         Ensure.Always.ArgumentNotNull(array, nameof(array));
132         Ensure.Always.ArgumentInRange(arrayIndex, new
133             ↳ Range<int>(0, array.Length - 1), nameof(arrayIndex));
134         if (arrayIndex + Length > array.Length)
135         {
136             throw new InvalidOperationException();
137         }
138         array[arrayIndex++] = Index;
139         array[arrayIndex++] = Source;
140         array[arrayIndex] = Target;
141     }
142
143     public bool Remove(TLink item) =>
144     ↳ Throw.A.NotSupportedExceptionAndReturn<bool>();
145
146     public int IndexOf(TLink item)
147     {
148         if (_equalityComparer.Equals(Index, item))
149         {
150             return _constants.IndexPart;
151         }
152         if (_equalityComparer.Equals(Source, item))
153         {
154             return _constants.SourcePart;
155         }
156         if (_equalityComparer.Equals(Target, item))
157         {
158             return _constants.TargetPart;
159         }
160         return -1;
161     }
162
163     public void Insert(int index, TLink item) => throw new
164     ↳ NotSupportedException();
165
166     public void RemoveAt(int index) => throw new
167     ↳ NotSupportedException();
168
169     #endregion
170 }

```

./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>
8         ↳ link) => 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 =
7         ↳ links;
8     }

```

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

```

1 //-----
2 ↳ -----
3 // <auto-generated>
4 //     Generated by the MSBuild WriteCodeFragment class.
5 // </auto-generated>
6 ↳ -----
7
8 using System;
9 using System.Reflection;
10
11 [assembly: System.Reflection.AssemblyConfigurationAttribute("Debug")]
12 [assembly: System.Reflection.AssemblyCopyrightAttribute("Konstantin
13 ↳ Diachenko")]
14 [assembly:
15 ↳ System.Reflection.AssemblyDescriptionAttribute("LinksPlatform\'s
16 ↳ Platform.Data.Doublets Class Library")]
17 [assembly: System.Reflection.AssemblyFileVersionAttribute("0.0.1.0")]
18 [assembly:
19 ↳ System.Reflection.AssemblyInformationalVersionAttribute("0.0.1")]
20 [assembly:
21 ↳ System.Reflection.AssemblyTitleAttribute("Platform.Data.Doublets")]
22 [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> :
8     ↳ LinksOperatorBase<TLink>, IPropertyOperator<TLink, TLink>
9     {
10         private static readonly EqualityComparer<TLink>
11         ↳ _equalityComparer = EqualityComparer<TLink>.Default;
12
13         public DefaultLinkPropertyOperator(ILinks<TLink> links) :
14         ↳ base(links)
15         {
16         }
17
18         public TLink GetValue(TLink @object, TLink property)
19         {
20             var objectProperty = Links.SearchOrDefault(@object,
21             ↳ property);
22             if (_equalityComparer.Equals(objectProperty, default))
23             {
24                 return default;
25             }
26         }
27     }

```

```

22     var valueLink = Links.All(Links.Constants.Any,
23         ↪ objectProperty).SingleOrDefault();
24     if (valueLink == null)
25     {
26         return default;
27     }
28     var value =
29         ↪ Links.GetTarget(valueLink[Links.Constants.IndexPart]);
30     return value;
31 }
32
33 public void SetValue(TLink @object, TLink property, TLink
34     ↪ value)
35 {
36     var objectProperty = Links.GetOrCreate(@object, property);
37     Links.DeleteMany(Links.All(Links.Constants.Any,
38         ↪ objectProperty).Select(link =>
39         ↪ link[Links.Constants.IndexPart]).ToList());
40     Links.GetOrCreate(objectProperty, value);
41 }
42 }
43 }

```

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

```

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

```

```

59         public static TLink GetSource(IntPtr pointer) => (pointer
60             ↳ + SourceOffset).GetValue<TLink>();
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         public static TLink GetTarget(IntPtr pointer) => (pointer
63             ↳ + TargetOffset).GetValue<TLink>();
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         public static TLink GetLeftAsSource(IntPtr pointer) =>
66             ↳ (pointer + LeftAsSourceOffset).GetValue<TLink>();
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public static TLink GetRightAsSource(IntPtr pointer) =>
69             ↳ (pointer + RightAsSourceOffset).GetValue<TLink>();
70         [MethodImpl(MethodImplOptions.AggressiveInlining)]
71         public static TLink GetSizeAsSource(IntPtr pointer) =>
72             ↳ (pointer + SizeAsSourceOffset).GetValue<TLink>();
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         public static TLink GetLeftAsTarget(IntPtr pointer) =>
75             ↳ (pointer + LeftAsTargetOffset).GetValue<TLink>();
76         [MethodImpl(MethodImplOptions.AggressiveInlining)]
77         public static TLink GetRightAsTarget(IntPtr pointer) =>
78             ↳ (pointer + RightAsTargetOffset).GetValue<TLink>();
79         [MethodImpl(MethodImplOptions.AggressiveInlining)]
80         public static TLink GetSizeAsTarget(IntPtr pointer) =>
81             ↳ (pointer + SizeAsTargetOffset).GetValue<TLink>();
82
83         [MethodImpl(MethodImplOptions.AggressiveInlining)]
84         public static void SetSource(IntPtr pointer, TLink value)
85             ↳ => (pointer + SourceOffset).SetValue(value);
86         [MethodImpl(MethodImplOptions.AggressiveInlining)]
87         public static void SetTarget(IntPtr pointer, TLink value)
88             ↳ => (pointer + TargetOffset).SetValue(value);
89         [MethodImpl(MethodImplOptions.AggressiveInlining)]
90         public static void SetLeftAsSource(IntPtr pointer, TLink
91             ↳ value) => (pointer +
92             ↳ LeftAsSourceOffset).SetValue(value);
93         [MethodImpl(MethodImplOptions.AggressiveInlining)]
94         public static void SetRightAsSource(IntPtr pointer, TLink
95             ↳ value) => (pointer +
96             ↳ RightAsSourceOffset).SetValue(value);
97         [MethodImpl(MethodImplOptions.AggressiveInlining)]
98         public static void SetSizeAsSource(IntPtr pointer, TLink
99             ↳ value) => (pointer +
100             ↳ SizeAsSourceOffset).SetValue(value);
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         public static void SetLeftAsTarget(IntPtr pointer, TLink
103             ↳ value) => (pointer +
104             ↳ LeftAsTargetOffset).SetValue(value);
105         [MethodImpl(MethodImplOptions.AggressiveInlining)]
106         public static void SetRightAsTarget(IntPtr pointer, TLink
107             ↳ value) => (pointer +
108             ↳ RightAsTargetOffset).SetValue(value);
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         public static void SetSizeAsTarget(IntPtr pointer, TLink
111             ↳ value) => (pointer +
112             ↳ SizeAsTargetOffset).SetValue(value);
113
114     }
115
116     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
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
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
    }

    private readonly long _memoryReservationStep;

    private readonly IResizableDirectMemory _memory;
    private IntPtr _header;
    private IntPtr _links;

    private LinksTargetsTreeMethods _targetsTreeMethods;
    private LinksSourcesTreeMethods _sourcesTreeMethods;

    // 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
    private UnusedLinksListMethods _unusedLinksListMethods;

    /// <summary>
    /// Возвращает общее число связей находящихся в хранилище.
    /// </summary>
    private TLink Total =>
        ↳ Subtract(LinksHeader.GetAllocatedLinks(_header),
        ↳ LinksHeader.GetFreeLinks(_header));

    public LinksCombinedConstants<TLink, TLink, int> Constants {
        ↳ get; }

    public ResizableDirectMemoryLinks(string address)
        : this(address, DefaultLinksSizeStep)

```

```

169 {
170 }
171
172 /// <summary>
173 /// Создаёт экземпляр базы данных Links в файле по указанному
174   ↳ адресу, с указанным минимальным шагом расширения базы
175   ↳ данных.
176 /// </summary>
177 /// <param name="address">Полный путь к файлу базы
178   ↳ данных.</param>
179 /// <param name="memoryReservationStep">Минимальный шаг
180   ↳ расширения базы данных в байтах.</param>
181 public ResizableDirectMemoryLinks(string address, long
182   ↳ memoryReservationStep)
183   : this(new FileMappedResizableDirectMemory(address,
184   ↳ memoryReservationStep), memoryReservationStep)
185 {
186 }
187
188 public ResizableDirectMemoryLinks(IResizableDirectMemory
189   ↳ memory)
190   : this(memory, DefaultLinksSizeStep)
191 {
192 }
193
194 public ResizableDirectMemoryLinks(IResizableDirectMemory
195   ↳ memory, long memoryReservationStep)
196 {
197     Constants = Default<LinksCombinedConstants<TLink, TLink,
198   ↳ int>>.Instance;
199     _memory = memory;
200     _memoryReservationStep = memoryReservationStep;
201     if (memory.ReservedCapacity < memoryReservationStep)
202     {
203         memory.ReservedCapacity = memoryReservationStep;
204     }
205     SetPointers(_memory);
206     // Гарантия корректности _memory.UsedCapacity относительно
207     // _header->AllocatedLinks
208     _memory.UsedCapacity = ((long)(Integer<TLink>)LinksHeader.
209   ↳ GetAllocatedLinks(_header) * LinkSizeInBytes) +
210   ↳ LinkHeaderSizeInBytes;
211     // Гарантия корректности _header->ReservedLinks
212     // относительно _memory.ReservedCapacity
213     LinksHeader.SetReservedLinks(_header,
214   ↳ (Integer<TLink>)((_memory.ReservedCapacity -
215   ↳ LinkHeaderSizeInBytes) / LinkSizeInBytes));
216 }
217
218 [MethodImpl(MethodImplOptions.AggressiveInlining)]
219 public TLink Count(IList<TLink> restrictions)
220 {
221     // Если нет ограничений, тогда возвращаем общее число
222     // связей находящихся в хранилище.
223     if (restrictions.Count == 0)
224     {
225         return Total;
226     }
227     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 :
219   ↳ Integer<TLink>.Zero;
220 }
221 if (restrictions.Count == 2)
222 {
223     var index = restrictions[Constants.IndexPart];
224     var value = restrictions[1];
225     if (_equalityComparer.Equals(index, Constants.Any))
226     {
227         if (_equalityComparer.Equals(value, Constants.Any))
228         {
229             return Total; // Any - как отсутствие
230   ↳ ограничения
231         }
232         return Add(_sourcesTreeMethods.CalculateReferences
233   ↳ (value),
234   ↳ _targetsTreeMethods.CalculateReferences(value)
235   ↳ );
236     }
237     else
238     {
239         if (!Exists(index))
240         {
241             return Integer<TLink>.Zero;
242         }
243         if (_equalityComparer.Equals(value, Constants.Any))
244         {
245             return Integer<TLink>.One;
246         }
247         var storedLinkValue = GetLinkUnsafe(index);
248         if (_equalityComparer.Equals(Link.GetSource(stored
249   ↳ LinkValue), value)
250   ↳ ||
251   ↳ _equalityComparer.Equals(Link.GetTarget(stored
252   ↳ LinkValue),
253   ↳ value))
254         {
255             return Integer<TLink>.One;
256         }
257         return Integer<TLink>.Zero;
258     }
259 }
260 if (restrictions.Count == 3)
261 {
262     var index = restrictions[Constants.IndexPart];
263     var source = restrictions[Constants.SourcePart];
264     var target = restrictions[Constants.TargetPart];
265
266     if (_equalityComparer.Equals(index, Constants.Any))
267     {

```

```

259         if (_equalityComparer.Equals(source,
260             ↪ Constants.Any) &&
261             ↪ _equalityComparer.Equals(target,
262             ↪ Constants.Any))
263         {
264             return Total;
265         }
266     else if (_equalityComparer.Equals(source,
267         ↪ Constants.Any))
268     {
269         return _targetsTreeMethods.CalculateReferences
270             ↪ (target);
271     }
272     else if (_equalityComparer.Equals(target,
273         ↪ Constants.Any))
274     {
275         return _sourcesTreeMethods.CalculateReferences
276             ↪ (source);
277     }
278     else //if(source != Any && target != Any)
279     {
280         // Эквивалент Exists(source, target) =>
281         ↪ Count(Any, source, target) > 0
282         var link = _sourcesTreeMethods.Search(source,
283             ↪ target);
284         return _equalityComparer.Equals(link,
285             ↪ Constants.Null) ? Integer<TLink>.Zero :
286             ↪ Integer<TLink>.One;
287     }
288 }
289 else
290 {
291     if (!Exists(index))
292     {
293         return Integer<TLink>.Zero;
294     }
295     if (_equalityComparer.Equals(source,
296         ↪ Constants.Any) &&
297         ↪ _equalityComparer.Equals(target,
298         ↪ Constants.Any))
299     {
300         return Integer<TLink>.One;
301     }
302     var storedLinkValue = GetLinkUnsafe(index);
303     if (!_equalityComparer.Equals(source,
304         ↪ Constants.Any) &&
305         ↪ !_equalityComparer.Equals(target,
306         ↪ Constants.Any))
307     {
308         if (_equalityComparer.Equals(Link.GetSource(st
309             ↪ oredLinkValue), source)
310             ↪ &&
311             ↪ _equalityComparer.Equals(Link.GetTarget(st
312             ↪ oredLinkValue),
313             ↪ target))
314         {
315             return Integer<TLink>.One;
316         }
317     }
318 }
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

```

341     {
342         return Constants.Continue;
343     }
344     return handler(GetLinkStruct(index));
345 }
346 if (restrictions.Count == 2)
347 {
348     var index = restrictions[Constants.IndexPart];
349     var value = restrictions[1];
350     if (_equalityComparer.Equals(index, Constants.Any))
351     {
352         if (_equalityComparer.Equals(value, Constants.Any))
353         {
354             return Each(handler, ArrayPool<TLink>.Empty);
355         }
356         if (_equalityComparer.Equals(Each(handler, new[] {
357             ↪ index, value, Constants.Any })),
358             ↪ Constants.Break))
359         {
360             return Constants.Break;
361         }
362         return Each(handler, new[] { index, Constants.Any,
363             ↪ value });
364     }
365     else
366     {
367         if (!Exists(index))
368         {
369             return Constants.Continue;
370         }
371         if (_equalityComparer.Equals(value, Constants.Any))
372         {
373             return handler(GetLinkStruct(index));
374         }
375         var storedLinkValue = GetLinkUnsafe(index);
376         if (_equalityComparer.Equals(Link.GetSource(stored
377             ↪ LinkValue), value)
378             ↪ ||
379             ↪ _equalityComparer.Equals(Link.GetTarget(stored
380             ↪ LinkValue),
381             ↪ value))
382         {
383             return handler(GetLinkStruct(index));
384         }
385         return Constants.Continue;
386     }
387 }
388 if (restrictions.Count == 3)
389 {
390     var index = restrictions[Constants.IndexPart];
391     var source = restrictions[Constants.SourcePart];
392     var target = restrictions[Constants.TargetPart];
393     if (_equalityComparer.Equals(index, Constants.Any))
394     {
395         if (_equalityComparer.Equals(source,
396             ↪ Constants.Any) &&
397             ↪ _equalityComparer.Equals(target,
398             ↪ Constants.Any))
399         {
400             return Each(handler, ArrayPool<TLink>.Empty);
401         }
402         else if (_equalityComparer.Equals(source,
403             ↪ Constants.Any))
404         {
405             return
406                 ↪ _targetsTreeMethods.EachReference(target,
407                 ↪ handler);
408         }
409         else if (_equalityComparer.Equals(target,
410             ↪ Constants.Any))
411         {
412             return
413                 ↪ _sourcesTreeMethods.EachReference(source,
414                 ↪ handler);
415         }
416         else //if(source != Any && target != Any)
417         {
418             var link = _sourcesTreeMethods.Search(source,
419                 ↪ target);
420             return _equalityComparer.Equals(link,
421                 ↪ Constants.Null) ? Constants.Continue :
422                 ↪ handler(GetLinkStruct(link));
423         }
424     }
425     }
426     else
427     {
428         if (!Exists(index))
429         {
430             return Constants.Continue;
431         }
432         if (_equalityComparer.Equals(source,
433             ↪ Constants.Any) &&
434             ↪ _equalityComparer.Equals(target,
435             ↪ Constants.Any))
436         {
437             return handler(GetLinkStruct(index));
438         }
439         var storedLinkValue = GetLinkUnsafe(index);
440         if (!_equalityComparer.Equals(source,
441             ↪ Constants.Any) &&
442             ↪ !_equalityComparer.Equals(target,
443             ↪ Constants.Any))
444         {
445             if (_equalityComparer.Equals(Link.GetSource(st
446                 ↪ oredLinkValue), source)
447                 ↪ &&
448                 ↪ _equalityComparer.Equals(Link.GetTarget(st
449                 ↪ oredLinkValue),
450                 ↪ target))
451             {
452                 return handler(GetLinkStruct(index));
453             }
454             return Constants.Continue;
455         }
456         var value = default(TLink);
457         if (_equalityComparer.Equals(source,
458             ↪ Constants.Any))

```

```

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

```

```

468     }
469     if (!_equalityComparer.Equals(Link.GetTarget(link),
470     ↪ Constants.Null))
471     {
472         _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTarget ↪
473             ↪ tPointer(_header),
474             ↪ linkIndex);
475     }
476     return linkIndex;
477 }
478
479 [MethodImpl(MethodImplOptions.AggressiveInlining)]
480 public Link<TLink> GetLinkStruct(TLink linkIndex)
481 {
482     var link = GetLinkUnsafe(linkIndex);
483     return new Link<TLink>(linkIndex, Link.GetSource(link),
484     ↪ Link.GetTarget(link));
485 }
486
487 [MethodImpl(MethodImplOptions.AggressiveInlining)]
488 private IntPtr GetLinkUnsafe(TLink linkIndex) =>
489     ↪ _links.GetElement(LinkSizeInBytes, linkIndex);
490
491 /// <remarks>
492 /// TODO: Возможно нужно будет заполнение нулями, если внешнее
493 ↪ API ими не заполняет пространство
494 /// </remarks>
495 public TLink Create()
496 {
497     var freeLink = LinksHeader.GetFirstFreeLink(_header);
498     if (!_equalityComparer.Equals(freeLink, Constants.Null))
499     {
500         _unusedLinksListMethods.Detach(freeLink);
501     }
502     else
503     {
504         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_h ↪
505             ↪ eader), Constants.MaxPossibleIndex) >
506             ↪ 0)
507         {
508             throw new LinksLimitReachedException((Integer<TLink ↪
509             ↪ k>)Constants.MaxPossibleIndex);
510         }
511         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_h ↪
512             ↪ eader),
513             ↪ Decrement(LinksHeader.GetReservedLinks(_header)))
514             ↪ >= 0)
515         {
516             _memory.ReservedCapacity += _memory.ReservationStep;
517             SetPointers(_memory);
518             LinksHeader.SetReservedLinks(_header,
519             ↪ (Integer<TLink>)(_memory.ReservedCapacity /
520             ↪ LinkSizeInBytes));
521         }
522         LinksHeader.SetAllocatedLinks(_header,
523             ↪ Increment(LinksHeader.GetAllocatedLinks(_header)));
524         _memory.UsedCapacity += LinkSizeInBytes;
525         freeLink = LinksHeader.GetAllocatedLinks(_header);

```

```

511     }
512     return freeLink;
513 }
514
515 public void Delete(TLink link)
516 {
517     if (_comparer.Compare(link,
518         ↪ LinksHeader.GetAllocatedLinks(_header)) < 0)
519     {
520         _unusedLinksListMethods.AttachAsFirst(link);
521     }
522     else if (_equalityComparer.Equals(link,
523         ↪ LinksHeader.GetAllocatedLinks(_header)))
524     {
525         LinksHeader.SetAllocatedLinks(_header,
526             ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
527         _memory.UsedCapacity -= LinkSizeInBytes;
528         // Убираем все связи, находящиеся в списке свободных в
529         ↪ конце файла, до тех пор, пока не дойдём до первой
530         ↪ существующей связи
531         // Позволяет оптимизировать количество выделенных
532         ↪ связей (AllocatedLinks)
533         while ((_comparer.Compare(LinksHeader.GetAllocatedLink
534             ↪ s(_header), Integer<TLink>.Zero) > 0) &&
535             ↪ IsUnusedLink(LinksHeader.GetAllocatedLinks(_header
536             ↪ )))
537         {
538             _unusedLinksListMethods.Detach(LinksHeader.GetAllo
539             ↪ catedLinks(_header));
540             LinksHeader.SetAllocatedLinks(_header, Decrement(L
541             ↪ inksHeader.GetAllocatedLinks(_header)));
542             _memory.UsedCapacity -= LinkSizeInBytes;
543         }
544     }
545 }
546
547 /// <remarks>
548 /// TODO: Возможно это должно быть событием, вызываемым из
549 ↪ IMemory, в том случае, если адрес реально поменялся
550 ///
551 /// Указатель this.links может быть в том же месте,
552 /// так как 0-я связь не используется и имеет такой же размер
553 ↪ как Header,
554 /// поэтому header размещается в том же месте, что и 0-я связь
555 /// </remarks>
556 private void SetPointers(IDirectMemory memory)
557 {
558     if (memory == null)
559     {
560         _links = IntPtr.Zero;
561         _header = _links;
562         _unusedLinksListMethods = null;
563         _targetsTreeMethods = null;
564         _unusedLinksListMethods = null;
565     }
566     else
567     {
568         _links = memory.Pointer;
569         _header = _links;
570     }
571 }

```

```

557     _sourcesTreeMethods = new
558     ↪ LinksSourcesTreeMethods(this);
559     _targetsTreeMethods = new
560     ↪ LinksTargetsTreeMethods(this);
561     _unusedLinksListMethods = new
562     ↪ UnusedLinksListMethods(_links, _header);
563 }
564
565 [MethodImpl(MethodImplOptions.AggressiveInlining)]
566 private bool Exists(TLink link)
567 => (_comparer.Compare(link, Constants.MinPossibleIndex) >=
568     ↪ 0)
569     && (_comparer.Compare(link,
570         ↪ LinksHeader.GetAllocatedLinks(_header)) <= 0)
571     && !IsUnusedLink(link);
572
573 [MethodImpl(MethodImplOptions.AggressiveInlining)]
574 private bool IsUnusedLink(TLink link)
575 => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_
576     ↪ header),
577     ↪ link)
578     || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkU
579     ↪ nsafe(link)),
580         ↪ Constants.Null)
581     && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(
582     ↪ link)),
583         ↪ Constants.Null));
584
585 #region DisposableBase
586
587 protected override bool AllowMultipleDisposeCalls => true;
588
589 protected override void DisposeCore(bool manual, bool
590     ↪ wasDisposed)
591 {
592     if (!wasDisposed)
593     {
594         SetPointers(null);
595     }
596     Disposable.TryDispose(_memory);
597 }
598
599 #endregion
600 }

```

./ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs

```

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

```

```

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 +
    ↳ LinksHeader.FirstFreeLinkOffset).GetValue<TLink>();
21
22     protected override TLink GetLast() => (_header +
    ↳ LinksHeader.LastFreeLinkOffset).GetValue<TLink>();
23
24     protected override TLink GetPrevious(TLink element) =>
    ↳ (_links.GetElement(LinkSizeInBytes, element) +
    ↳ Link.SourceOffset).GetValue<TLink>();
25
26     protected override TLink GetNext(TLink element) =>
    ↳ (_links.GetElement(LinkSizeInBytes, element) +
    ↳ Link.TargetOffset).GetValue<TLink>();
27
28     protected override TLink GetSize() => (_header +
    ↳ LinksHeader.FreeLinksOffset).GetValue<TLink>();
29
30     protected override void SetFirst(TLink element) =>
    ↳ (_header +
    ↳ LinksHeader.FirstFreeLinkOffset).SetValue(element);
31
32     protected override void SetLast(TLink element) => (_header
    ↳ + LinksHeader.LastFreeLinkOffset).SetValue(element);
33
34     protected override void SetPrevious(TLink element, TLink
    ↳ previous) => (_links.GetElement(LinkSizeInBytes,
    ↳ element) + Link.SourceOffset).SetValue(previous);
35
36     protected override void SetNext(TLink element, TLink next)
    ↳ => (_links.GetElement(LinkSizeInBytes, element) +
    ↳ Link.TargetOffset).SetValue(next);
37
38     protected override void SetSize(TLink size) => (_header +
    ↳ LinksHeader.FreeLinksOffset).SetValue(size);
39
40 }
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 :
    ↳ SizedAndThreadedAVLBalancedTreeMethods<TLink>
15     {
16         private readonly ResizableDirectMemoryLinks<TLink> _memory;
17         private readonly LinksCombinedConstants<TLink, TLink, int>
    ↳ _constants;
18         protected readonly IntPtr Links;
19         protected readonly IntPtr Header;
20
21     protected
    ↳ LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink>
    ↳ memory)
    {
22         Links = memory._links;
23         Header = memory._header;
24         _memory = memory;
25         _constants = memory.Constants;
26     }
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected abstract TLink GetTreeRoot();
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected abstract TLink GetBasePartValue(TLink link);
33
34     public TLink this[TLink index]
35     {
36         get
37         {
38             var root = GetTreeRoot();
39             if (GreaterOrEqualThan(index, GetSize(root)))
40             {
41                 return GetZero();
42             }
43             while (!EqualToZero(root))
44             {
45                 var left = GetLeftOrDefault(root);
46                 var leftSize = GetSizeOrZero(left);
47                 if (LessThan(index, leftSize))
48                 {
49                     root = left;
50                     continue;
51                 }
52                 if (IsEquals(index, leftSize))
53                 {
54                     return root;
55                 }
56                 root = GetRightOrDefault(root);
57                 index = Subtract(index, Increment(leftSize));
58             }
59             return GetZero(); // TODO: Impossible situation
60             ↳ 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,
80                                     ↪ Increment(GetRightSize(root)));
81             root = GetLeftOrDefault(root);
82         }
83     }
84     root = GetTreeRoot();
85     var totalLeftIgnore = GetZero();
86     while (!EqualToZero(root))
87     {
88         var @base = GetBasePartValue(root);
89         if (GreaterOrEqualThan(@base, link))
90         {
91             root = GetLeftOrDefault(root);
92         }
93         else
94         {
95             totalLeftIgnore = Add(totalLeftIgnore,
96                                     ↪ Increment(GetLeftSize(root)));
97             root = GetRightOrDefault(root);
98         }
99     }
100     return Subtract(Subtract(total, totalRightIgnore),
101                     ↪ totalLeftIgnore);
102 }
103
104 public TLink EachReference(TLink link, Func<IList<TLink>,
105 ↪ TLink> handler)
106 {
107     var root = GetTreeRoot();
108     if (EqualToZero(root))
109     {
110         return _constants.Continue;
111     }
112     TLink first = GetZero(), current = root;
113     while (!EqualToZero(current))
114     {
115         var @base = GetBasePartValue(current);
116         if (GreaterOrEqualThan(@base, link))
117         {
118             if (IsEquals(@base, link))
119             {
120                 first = current;
121             }
122             current = GetLeftOrDefault(current);
123         }
124         else
125         {
126             current = GetRightOrDefault(current);

```

```

124     }
125 }
126 if (!EqualToZero(first))
127 {
128     current = first;
129     while (true)
130     {
131         if (IsEquals(handler(_memory.GetLinkStruct(current),
132 ↪ rent)),
133 ↪ _constants.Break))
134         {
135             return _constants.Break;
136         }
137         current = GetNext(current);
138         if (EqualToZero(current) ||
139 ↪ !IsEquals(GetBasePartValue(current), link))
140         {
141             break;
142         }
143     }
144 }
145 return _constants.Continue;
146 }
147
148 protected override void PrintNodeValue(TLink node,
149 ↪ StringBuilder sb)
150 {
151     sb.Append(' ');
152     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
153 ↪ Link.SourceOffset).GetValue<TLink>());
154     sb.Append(' - ');
155     sb.Append(' > ');
156     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
157 ↪ Link.TargetOffset).GetValue<TLink>());
158 }
159 }
160
161 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
162 {
163     public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<
164 ↪ TLink>
165 ↪ memory)
166     : base(memory)
167     {
168     }
169 }
170
171 protected override IntPtr GetLeftPointer(TLink node) =>
172     ↪ Links.GetElement(LinkSizeInBytes, node) +
173     ↪ Link.LeftAsSourceOffset;
174
175 protected override IntPtr GetRightPointer(TLink node) =>
176     ↪ Links.GetElement(LinkSizeInBytes, node) +
177     ↪ Link.RightAsSourceOffset;
178
179 protected override TLink GetLeftValue(TLink node) =>
180     ↪ (Links.GetElement(LinkSizeInBytes, node) +
181     ↪ Link.LeftAsSourceOffset).GetValue<TLink>();

```

```

168     protected override TLink GetRightValue(TLink node) =>
169         ↪ (Links.GetElement(LinkSizeInBytes, node) +
170         ↪ Link.RightAsSourceOffset).GetValue<TLink>();
171
172     protected override TLink GetSize(TLink node)
173     {
174         var previousValue = (Links.GetElement(LinkSizeInBytes,
175         ↪ node) + Link.SizeAsSourceOffset).GetValue<TLink>());
176         return BitwiseHelpers.PartialRead(previousValue, 5,
177         ↪ -5);
178     }
179
180     protected override void SetLeft(TLink node, TLink left) =>
181         ↪ (Links.GetElement(LinkSizeInBytes, node) +
182         ↪ Link.LeftAsSourceOffset).SetValue(left);
183
184     protected override void SetRight(TLink node, TLink right)
185         ↪ => (Links.GetElement(LinkSizeInBytes, node) +
186         ↪ Link.RightAsSourceOffset).SetValue(right);
187
188     protected override void SetSize(TLink node, TLink size)
189     {
190         var previousValue = (Links.GetElement(LinkSizeInBytes,
191         ↪ node) + Link.SizeAsSourceOffset).GetValue<TLink>());
192         (Links.GetElement(LinkSizeInBytes, node) +
193         ↪ Link.SizeAsSourceOffset).SetValue(BitwiseHelpers.P
194         ↪ artialWrite(previousValue, size, 5,
195         ↪ -5));
196     }
197
198     protected override bool GetLeftIsChild(TLink node)
199     {
200         var previousValue = (Links.GetElement(LinkSizeInBytes,
201         ↪ node) + Link.SizeAsSourceOffset).GetValue<TLink>());
202         return (Integer<TLink>)BitwiseHelpers.PartialRead(prev
203         ↪ iousValue, 4,
204         ↪ 1);
205     }
206
207     protected override void SetLeftIsChild(TLink node, bool
208         ↪ value)
209     {
210         var previousValue = (Links.GetElement(LinkSizeInBytes,
211         ↪ node) + Link.SizeAsSourceOffset).GetValue<TLink>());
212         var modified =
213         ↪ BitwiseHelpers.PartialWrite(previousValue,
214         ↪ (TLink)(Integer<TLink>)value, 4, 1);
215         (Links.GetElement(LinkSizeInBytes, node) +
216         ↪ Link.SizeAsSourceOffset).SetValue(modified);
217     }
218
219     protected override bool GetRightIsChild(TLink node)
220     {
221         var previousValue = (Links.GetElement(LinkSizeInBytes,
222         ↪ node) + Link.SizeAsSourceOffset).GetValue<TLink>());
223         return (Integer<TLink>)BitwiseHelpers.PartialRead(prev
224         ↪ iousValue, 3,
225         ↪ 1);
226     }

```

```

203     }
204
205     protected override void SetRightIsChild(TLink node, bool
206         ↪ value)
207     {
208         var previousValue = (Links.GetElement(LinkSizeInBytes,
209         ↪ node) + Link.SizeAsSourceOffset).GetValue<TLink>());
210         var modified =
211         ↪ BitwiseHelpers.PartialWrite(previousValue,
212         ↪ (TLink)(Integer<TLink>)value, 3, 1);
213         (Links.GetElement(LinkSizeInBytes, node) +
214         ↪ Link.SizeAsSourceOffset).SetValue(modified);
215     }
216
217     protected override sbyte GetBalance(TLink node)
218     {
219         var previousValue = (Links.GetElement(LinkSizeInBytes,
220         ↪ node) + Link.SizeAsSourceOffset).GetValue<TLink>());
221         var value = (ulong)(Integer<TLink>)BitwiseHelpers.Part
222         ↪ ialRead(previousValue, 0,
223         ↪ 3);
224         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value
225         ↪ & 4) << 5) | value & 3 | 124 : value & 3);
226         return unpackedValue;
227     }
228
229     protected override void SetBalance(TLink node, sbyte value)
230     {
231         var previousValue = (Links.GetElement(LinkSizeInBytes,
232         ↪ node) + Link.SizeAsSourceOffset).GetValue<TLink>());
233         var packagedValue =
234         ↪ (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) |
235         ↪ value & 3);
236         var modified =
237         ↪ BitwiseHelpers.PartialWrite(previousValue,
238         ↪ packagedValue, 0, 3);
239         (Links.GetElement(LinkSizeInBytes, node) +
240         ↪ Link.SizeAsSourceOffset).SetValue(modified);
241     }
242
243     protected override bool FirstIsToTheLeftOfSecond(TLink
244         ↪ first, TLink second)
245     {
246         var firstSource = (Links.GetElement(LinkSizeInBytes,
247         ↪ first) + Link.SourceOffset).GetValue<TLink>());
248         var secondSource = (Links.GetElement(LinkSizeInBytes,
249         ↪ second) + Link.SourceOffset).GetValue<TLink>());
250         return LessThan(firstSource, secondSource) ||
251         ↪ (IsEquals(firstSource, secondSource) &&
252         ↪ LessThan((Links.GetElement(LinkSizeInBytes,
253         ↪ first) +
254         ↪ Link.TargetOffset).GetValue<TLink>(),
255         ↪ (Links.GetElement(LinkSizeInBytes, second)
256         ↪ + Link.TargetOffset).GetValue<TLink>()));
257     }

```

```

236     protected override bool FirstIsToTheRightOfSecond(TLink
    ↪ first, TLink second)
237 {
238     var firstSource = (Links.GetElement(LinkSizeInBytes,
    ↪ first) + Link.SourceOffset).GetValue<TLink>();
239     var secondSource = (Links.GetElement(LinkSizeInBytes,
    ↪ second) + Link.SourceOffset).GetValue<TLink>();
240     return GreaterThan(firstSource, secondSource) ||
241         (IsEquals(firstSource, secondSource) && Greater
    ↪ Than((Links.GetElement(LinkSizeInBytes,
    ↪ first) +
    ↪ Link.TargetOffset).GetValue<TLink>(),
    ↪ (Links.GetElement(LinkSizeInBytes, second)
    ↪ + Link.TargetOffset).GetValue<TLink>()));
242 }
243
244 protected override TLink GetTreeRoot() => (Header +
    ↪ LinksHeader.FirstAsSourceOffset).GetValue<TLink>();
245
246 protected override TLink GetBasePartValue(TLink link) =>
    ↪ (Links.GetElement(LinkSizeInBytes, link) +
    ↪ Link.SourceOffset).GetValue<TLink>();
247
248 /// <summary>
249 /// Выполняет поиск и возвращает индекс связи с указанными
    ↪ Source (началом) и Target (концом)
250 /// по дереву (индексу) связей, отсортированному по
    ↪ Source, а затем по Target.
251 /// </summary>
252 /// <param name="source">Индекс связи, которая является
    ↪ началом на искомой связи.</param>
253 /// <param name="target">Индекс связи, которая является
    ↪ концом на искомой связи.</param>
254 /// <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) +
    ↪ Link.SourceOffset).GetValue<TLink>();
261         var rootTarget =
    ↪ (Links.GetElement(LinkSizeInBytes, root) +
    ↪ Link.TargetOffset).GetValue<TLink>();
262         if (FirstIsToTheLeftOfSecond(source, target,
    ↪ rootSource, rootTarget)) // node.Key < root.Key
263         {
264             root = GetLeftOrDefault(root);
265         }
266         else if (FirstIsToTheRightOfSecond(source, target,
    ↪ rootSource, rootTarget)) // node.Key > root.Key
267         {
268             root = GetRightOrDefault(root);
269         }
270         else // node.Key == root.Key
271         {
272             return root;

```

```

273     }
274 }
275     return GetZero();
276 }
277
278 [MethodImpl(MethodImplOptions.AggressiveInlining)]
279 private bool FirstIsToTheLeftOfSecond(TLink firstSource,
    ↪ TLink firstTarget, TLink secondSource, TLink
    ↪ secondTarget) => LessThan(firstSource, secondSource)
    ↪ || (IsEquals(firstSource, secondSource) &&
    ↪ LessThan(firstTarget, secondTarget));
280
281 [MethodImpl(MethodImplOptions.AggressiveInlining)]
282 private bool FirstIsToTheRightOfSecond(TLink firstSource,
    ↪ TLink firstTarget, TLink secondSource, TLink
    ↪ secondTarget) => GreaterThan(firstSource,
    ↪ secondSource) || (IsEquals(firstSource, secondSource)
    ↪ && GreaterThan(firstTarget, secondTarget));
283 }
284
285 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
286 {
287     public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<
    ↪ TLink>
    ↪ memory)
    ↪ : base(memory)
288     {
289     }
290
291     protected override IntPtr GetLeftPointer(TLink node) =>
    ↪ Links.GetElement(LinkSizeInBytes, node) +
    ↪ Link.LeftAsTargetOffset;
292
293     protected override IntPtr GetRightPointer(TLink node) =>
    ↪ Links.GetElement(LinkSizeInBytes, node) +
    ↪ Link.RightAsTargetOffset;
294
295     protected override TLink GetLeftValue(TLink node) =>
    ↪ (Links.GetElement(LinkSizeInBytes, node) +
    ↪ Link.LeftAsTargetOffset).GetValue<TLink>();
296
297     protected override TLink GetRightValue(TLink node) =>
    ↪ (Links.GetElement(LinkSizeInBytes, node) +
    ↪ Link.RightAsTargetOffset).GetValue<TLink>();
298
299     protected override TLink GetSize(TLink node)
300     {
301         var previousValue = (Links.GetElement(LinkSizeInBytes,
    ↪ node) + Link.SizeAsTargetOffset).GetValue<TLink>();
302         return BitwiseHelpers.PartialRead(previousValue, 5,
    ↪ -5);
303     }
304
305     protected override void SetLeft(TLink node, TLink left) =>
    ↪ (Links.GetElement(LinkSizeInBytes, node) +
    ↪ Link.LeftAsTargetOffset).SetValue(left);
306
307

```

```

308     protected override void SetRight(TLink node, TLink right)
309     ↪ => (Links.GetElement(LinkSizeInBytes, node) +
310     ↪ Link.RightAsTargetOffset).SetValue(right);
311
312     protected override void SetSize(TLink node, TLink size)
313     {
314         var previousValue = (Links.GetElement(LinkSizeInBytes,
315         ↪ node) + Link.SizeAsTargetOffset).GetValue<TLink>();
316         (Links.GetElement(LinkSizeInBytes, node) +
317         ↪ Link.SizeAsTargetOffset).SetValue(BitwiseHelpers.P
318         ↪ artialWrite(previousValue, size, 5,
319         ↪ -5));
320     }
321
322     protected override bool GetLeftIsChild(TLink node)
323     {
324         var previousValue = (Links.GetElement(LinkSizeInBytes,
325         ↪ node) + Link.SizeAsTargetOffset).GetValue<TLink>());
326         return (Integer<TLink>)BitwiseHelpers.PartialRead(prev
327         ↪ ousValue, 4,
328         ↪ 1);
329     }
330
331     protected override void SetLeftIsChild(TLink node, bool
332     ↪ value)
333     {
334         var previousValue = (Links.GetElement(LinkSizeInBytes,
335         ↪ node) + Link.SizeAsTargetOffset).GetValue<TLink>());
336         var modified =
337         ↪ BitwiseHelpers.PartialWrite(previousValue,
338         ↪ (TLink)(Integer<TLink>)value, 4, 1);
339         (Links.GetElement(LinkSizeInBytes, node) +
340         ↪ Link.SizeAsTargetOffset).SetValue(modified);
341     }
342
343     protected override bool GetRightIsChild(TLink node)
344     {
345         var previousValue = (Links.GetElement(LinkSizeInBytes,
346         ↪ node) + Link.SizeAsTargetOffset).GetValue<TLink>());
347         return (Integer<TLink>)BitwiseHelpers.PartialRead(prev
348         ↪ ousValue, 3,
349         ↪ 1);
350     }
351
352     protected override void SetRightIsChild(TLink node, bool
353     ↪ value)
354     {
355         var previousValue = (Links.GetElement(LinkSizeInBytes,
356         ↪ node) + Link.SizeAsTargetOffset).GetValue<TLink>());
357         var modified =
358         ↪ BitwiseHelpers.PartialWrite(previousValue,
359         ↪ (TLink)(Integer<TLink>)value, 3, 1);
360         (Links.GetElement(LinkSizeInBytes, node) +
361         ↪ Link.SizeAsTargetOffset).SetValue(modified);
362     }
363
364     protected override sbyte GetBalance(TLink node)
365     {
366         var previousValue = (Links.GetElement(LinkSizeInBytes,
367         ↪ node) + Link.SizeAsTargetOffset).GetValue<TLink>());
368         var packagedValue =
369         ↪ (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) |
370         ↪ value & 3);
371         var modified =
372         ↪ BitwiseHelpers.PartialWrite(previousValue,
373         ↪ packagedValue, 0, 3);
374         (Links.GetElement(LinkSizeInBytes, node) +
375         ↪ Link.SizeAsTargetOffset).SetValue(modified);
376     }
377
378     protected override bool FirstIsToTheLeftOfSecond(TLink
379     ↪ first, TLink second)
380     {
381         var firstTarget = (Links.GetElement(LinkSizeInBytes,
382         ↪ first) + Link.TargetOffset).GetValue<TLink>());
383         var secondTarget = (Links.GetElement(LinkSizeInBytes,
384         ↪ second) + Link.TargetOffset).GetValue<TLink>());
385         return LessThan(firstTarget, secondTarget) ||
386         ↪ (IsEquals(firstTarget, secondTarget) &&
387         ↪ LessThan((Links.GetElement(LinkSizeInBytes,
388         ↪ first) +
389         ↪ Link.SourceOffset).GetValue<TLink>(),
390         ↪ (Links.GetElement(LinkSizeInBytes, second)
391         ↪ + Link.SourceOffset).GetValue<TLink>()));
392     }
393
394     protected override bool FirstIsToTheRightOfSecond(TLink
395     ↪ first, TLink second)
396     {
397         var firstTarget = (Links.GetElement(LinkSizeInBytes,
398         ↪ first) + Link.TargetOffset).GetValue<TLink>());
399         var secondTarget = (Links.GetElement(LinkSizeInBytes,
400         ↪ second) + Link.TargetOffset).GetValue<TLink>());
401         return GreaterThan(firstTarget, secondTarget) ||
402         ↪ (IsEquals(firstTarget, secondTarget) && Greater
403         ↪ Than((Links.GetElement(LinkSizeInBytes,
404         ↪ first) +
405         ↪ Link.SourceOffset).GetValue<TLink>(),
406         ↪ (Links.GetElement(LinkSizeInBytes, second)
407         ↪ + Link.SourceOffset).GetValue<TLink>()));
408     }
409
410     protected override TLink GetTreeRoot() => (Header +
411     ↪ LinksHeader.FirstAsTargetOffset).GetValue<TLink>());

```

```

375         protected override TLink GetBasePartValue(TLink link) =>
376             ↳ (Links.GetElement(LinkSizeInBytes, link) +
377                 ↳ Link.TargetOffset).GetValue<TLink>();
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 :
23          ↳ DisposableBase, ILinks<id>
24      {
25          /// <summary>Возвращает размер одной связи в байтах.</summary>
26          /// <remarks>
27          ///     Используется только во вне класса, не рекомендуется
28          ///     использовать внутри.
29          ///     Так как во вне не обязательно будет доступен unsafe C#.
30          /// </remarks>
31          public static readonly int LinkSizeInBytes = sizeof(Link);
32
33          public static readonly long DefaultLinksSizeStep =
34              ↳ LinkSizeInBytes * 1024 * 1024;
35
36          private struct Link
37          {
38              public id Source;
39              public id Target;
40              public id LeftAsSource;
41              public id RightAsSource;
42              public id SizeAsSource;
43              public id LeftAsTarget;
44              public id RightAsTarget;
45              public id SizeAsTarget;
46          }
47
48          private struct LinksHeader
49          {
50              public id AllocatedLinks;
51              public id ReservedLinks;
52              public id FreeLinks;
53              public id FirstFreeLink;
54              public id FirstAsSource;

```

```

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,
67      // ↳ является ли связь удалённой, нужно использовать не список
68      // ↳ а дерево, так как так можно быстрее проверить на наличие
69      // ↳ связи внутри
70      private UnusedLinksListMethods _unusedLinksListMethods;
71
72      /// <summary>
73      ///     Возвращает общее число связей находящихся в хранилище.
74      /// </summary>
75      private id Total => _header->AllocatedLinks -
76          ↳ _header->FreeLinks;
77
78      // TODO: Дать возможность переопределять в конструкторе
79      public LinksCombinedConstants<id, id, int> Constants { get; }
80
81      public UInt64ResizableDirectMemoryLinks(string address) :
82          ↳ this(address, DefaultLinksSizeStep) { }
83
84      /// <summary>
85      ///     Создаёт экземпляр базы данных Links в файле по указанному
86      ///     адресу, с указанным минимальным шагом расширения базы
87      ///     данных.
88      /// </summary>
89      /// <param name="address">Полный путь к файлу базы
90      ///     данных.</param>
91      /// <param name="memoryReservationStep">Минимальный шаг
92      ///     расширения базы данных в байтах.</param>
93      public UInt64ResizableDirectMemoryLinks(string address, long
94          memoryReservationStep) : this(new
95          FileMappedResizableDirectMemory(address,
96          memoryReservationStep), memoryReservationStep) { }
97
98      public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory
99          ↳ memory) : this(memory, DefaultLinksSizeStep) { }
100
101      public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory
102          ↳ memory, long memoryReservationStep)
103      {
104          Constants = Default<LinksCombinedConstants<id, id,
105              ↳ int>>.Instance;
106          _memory = memory;
107          _memoryReservationStep = memoryReservationStep;
108          if (memory.ReservedCapacity < memoryReservationStep)
109          {
110              memory.ReservedCapacity = memoryReservationStep;
111          }

```

```

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)
    ↪ + _targetsTreeMethods.CalculateReferences(val
    ↪ ue);
132         }
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 &&
    ↪ storedLinkValue->Target == target)
193             {
194                 return 1;
195             }
196             return 0;
197         }
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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
219 public id Each(Func<IList<id>, id> handler, IList<id>
↳ restrictions)
220 {
221     if (restrictions.Count == 0)
222     {
223         for (id link = 1; link <= _header->AllocatedLinks;
↳ link++)
224         {
225             if (Exists(link))
226             {
227                 if (handler(GetLinkStruct(link)) ==
↳ Constants.Break)
228                 {
229                     return Constants.Break;
230                 }
231             }
232         }
233         return Constants.Continue;
234     }
235     if (restrictions.Count == 1)
236     {
237         var index = restrictions[Constants.IndexPart];
238         if (index == Constants.Any)
239         {
240             return Each(handler, ArrayPool<ulong>.Empty);
241         }
242         if (!Exists(index))
243         {
244             return Constants.Continue;
245         }
246         return handler(GetLinkStruct(index));
247     }
248     if (restrictions.Count == 2)
249     {
250         var index = restrictions[Constants.IndexPart];
251         var value = restrictions[1];
252         if (index == Constants.Any)
253         {
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 ||
↳ storedLinkValue->Target == value)
277         {
278             return handler(GetLinkStruct(index));
279         }
280         return Constants.Continue;
281     }
282 }
283 if (restrictions.Count == 3)
284 {
285     var index = restrictions[Constants.IndexPart];
286     var source = restrictions[Constants.SourcePart];
287     var target = restrictions[Constants.TargetPart];
288     if (index == Constants.Any)
289     {
290         if (source == Constants.Any && target ==
↳ Constants.Any)
291         {
292             return Each(handler, ArrayPool<ulong>.Empty);
293         }
294         else if (source == Constants.Any)
295         {
296             return
↳ _targetsTreeMethods.EachReference(target,
↳ handler);
297         }
298         else if (target == Constants.Any)
299         {
300             return
↳ _sourcesTreeMethods.EachReference(source,
↳ handler);
301         }
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 &&
            storedLinkValue->Target == target)
323         {
324             return handler(GetLinkStruct(index));
325         }
326         return Constants.Continue;
327     }
328     var value = default(id);
329     if (source == Constants.Any)
330     {
331         value = target;
332     }
333     if (target == Constants.Any)
334     {
335         value = source;
336     }
337     if (storedLinkValue->Source == value ||
        storedLinkValue->Target == value)
338     {
339         return handler(GetLinkStruct(index));
340     }
341     return Constants.Continue;
342 }
343 }
344 }
345 throw new NotSupportedException("Другие размеры и способы
346 ↳ 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) =>
    ↳ &_links[linkIndex];

```



```

404     /// <remarks>
405     /// TODO: Возможно нужно будет заполнение нулями, если внешнее
406     ↪ 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 >
418                 ↪ Constants.MaxPossibleIndex)
419             {
420                 throw new LinksLimitReachedException(Constants.Max ↪
421                     ↪ PossibleIndex);
422             }
423             if (_header->AllocatedLinks >= _header->ReservedLinks
424                 ↪ - 1)
425             {
426                 _memory.ReservedCapacity += _memoryReservationStep;
427                 SetPointers(_memory);
428                 _header->ReservedLinks =
429                     ↪ (id)(_memory.ReservedCapacity / sizeof(Link));
430             }
431             _header->AllocatedLinks++;
432             _memory.UsedCapacity += sizeof(Link);
433             freeLink = _header->AllocatedLinks;
434         }
435         return freeLink;
436     }
437
438     public void Delete(id link)
439     {
440         if (link < _header->AllocatedLinks)
441         {
442             _unusedLinksListMethods.AttachAsFirst(link);
443         }
444         else if (link == _header->AllocatedLinks)
445         {
446             _header->AllocatedLinks--;
447             _memory.UsedCapacity -= sizeof(Link);
448             // Убираем все связи, находящиеся в списке свободных в
449             ↪ конце файла, до тех пор, пока не дойдём до первой
450             ↪ существующей связи
451             // Позволяет оптимизировать количество выделенных
452             ↪ связей (AllocatedLinks)
453             while (_header->AllocatedLinks > 0 &&
454                 ↪ IsUnusedLink(_header->AllocatedLinks))
455             {
456                 _unusedLinksListMethods.Detach(_header->AllocatedL ↪
457                     ↪ inks);
458                 _header->AllocatedLinks--;
459                 _memory.UsedCapacity -= sizeof(Link);
460             }
461         }
462     }

```

```

453     }
454
455     /// <remarks>
456     /// TODO: Возможно это должно быть событием, вызываемым из
457     ↪ IMemory, в том случае, если адрес реально поменялся
458     ///
459     /// Указатель this.links может быть в том же месте,
460     /// так как 0-я связь не используется и имеет такой же размер
461     ↪ как Header,
462     /// поэтому header размещается в том же месте, что и 0-я связь
463     /// </remarks>
464     private void SetPointers(IResizableDirectMemory memory)
465     {
466         if (memory == null)
467         {
468             _header = null;
469             _links = null;
470             _unusedLinksListMethods = null;
471             _targetsTreeMethods = null;
472             _unusedLinksListMethods = null;
473         }
474         else
475         {
476             _header = (LinksHeader*)(void*)memory.Pointer;
477             _links = (Link*)(void*)memory.Pointer;
478             _sourcesTreeMethods = new
479                 ↪ LinksSourcesTreeMethods(this);
480             _targetsTreeMethods = new
481                 ↪ LinksTargetsTreeMethods(this);
482             _unusedLinksListMethods = new
483                 ↪ UnusedLinksListMethods(_links, _header);
484         }
485     }
486
487     [MethodImpl(MethodImplOptions.AggressiveInlining)]
488     private bool Exists(id link) => link >=
489         ↪ Constants.MinPossibleIndex && link <=
490         ↪ _header->AllocatedLinks && !IsUnusedLink(link);
491
492     [MethodImpl(MethodImplOptions.AggressiveInlining)]
493     private bool IsUnusedLink(id link) => _header->FirstFreeLink
494         ↪ == link
495         || (_links[link].SizeAsSource
496             ↪ e == Constants.Null &&
497             ↪ _links[link].Source !=
498             ↪ Constants.Null);
499
500     #region Disposable
501
502     protected override bool AllowMultipleDisposeCalls => true;
503
504     protected override void DisposeCore(bool manual, bool
505         ↪ wasDisposed)
506     {
507         if (!wasDisposed)
508         {
509             SetPointers(null);
510         }
511         Disposable.TryDispose(_memory);
512     }

```

```

501     }
502     #endregion
503 }
504 }

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs
1  using Platform.Collections.Methods.Lists;
2
3  namespace Platform.Data.Doublets.ResizableDirectMemory
4  {
5      unsafe partial class UInt64ResizableDirectMemoryLinks
6      {
7          private class UnusedLinksListMethods :
8              ↳ CircularDoublyLinkedListMethods<ulong>
9          {
10             private readonly Link* _links;
11             private readonly LinksHeader* _header;
12
13             public UnusedLinksListMethods(Link* links, LinksHeader*
14                 ↳ header)
15             {
16                 _links = links;
17                 _header = header;
18             }
19
20             protected override ulong GetFirst() =>
21                 ↳ _header->FirstFreeLink;
22
23             protected override ulong GetLast() =>
24                 ↳ _header->LastFreeLink;
25
26             protected override ulong GetPrevious(ulong element) =>
27                 ↳ _links[element].Source;
28
29             protected override ulong GetNext(ulong element) =>
30                 ↳ _links[element].Target;
31
32             protected override ulong GetSize() => _header->FreeLinks;
33
34             protected override void SetFirst(ulong element) =>
35                 ↳ _header->FirstFreeLink = element;
36
37             protected override void SetLast(ulong element) =>
38                 ↳ _header->LastFreeLink = element;
39
40             protected override void SetPrevious(ulong element, ulong
41                 ↳ previous) => _links[element].Source = previous;
42
43             protected override void SetNext(ulong element, ulong next)
44                 ↳ => _links[element].Target = next;
45
46             protected override void SetSize(ulong size) =>
47                 ↳ _header->FreeLinks = size;
48         }
49     }
50 }

```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs

```

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

```

```

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

```

```

59     }
60 }
61
62 // TODO: Return indices range instead of references count
63 public ulong CalculateReferences(ulong link)
64 {
65     var root = GetTreeRoot();
66     var total = GetSize(root);
67     var totalRightIgnore = OUL;
68     while (root != 0)
69     {
70         var @base = GetBasePartValue(root);
71         if (@base <= link)
72         {
73             root = GetRightOrDefault(root);
74         }
75         else
76         {
77             totalRightIgnore += GetRightSize(root) + 1;
78             root = GetLeftOrDefault(root);
79         }
80     }
81     root = GetTreeRoot();
82     var totalLeftIgnore = OUL;
83     while (root != 0)
84     {
85         var @base = GetBasePartValue(root);
86         if (@base >= link)
87         {
88             root = GetLeftOrDefault(root);
89         }
90         else
91         {
92             totalLeftIgnore += GetLeftSize(root) + 1;
93             root = GetRightOrDefault(root);
94         }
95     }
96     return total - totalRightIgnore - totalLeftIgnore;
97 }
98
99 public ulong EachReference(ulong link, Func<IList<ulong>,
100     ↳ ulong> handler)
101 {
102     var root = GetTreeRoot();
103     if (root == 0)
104     {
105         return _constants.Continue;
106     }
107     ulong first = 0, current = root;
108     while (current != 0)
109     {
110         var @base = GetBasePartValue(current);
111         if (@base >= link)
112         {
113             if (@base == link)
114             {
115                 first = current;
116             }
117             current = GetLeftOrDefault(current);

```

```

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)) ==
129                 ↳ _constants.Break)
130             {
131                 return _constants.Break;
132             }
133             current = GetNext(current);
134             if (current == 0 || GetBasePartValue(current)
135                 ↳ != link)
136             {
137                 break;
138             }
139         }
140     }
141     return _constants.Continue;
142 }
143
144 protected override void PrintNodeValue(ulong node,
145     ↳ StringBuilder sb)
146 {
147     sb.Append(' ');
148     sb.Append(Links[node].Source);
149     sb.Append('-');
150     sb.Append('>');
151     sb.Append(Links[node].Target);
152 }
153
154 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
155 {
156     public LinksSourcesTreeMethods(UInt64ResizableDirectMemory
157     ↳ Links
158     ↳ memory)
159     : base(memory)
160     {
161     }
162
163     protected override IntPtr GetLeftPointer(ulong node) =>
164     ↳ new IntPtr(&Links[node].LeftAsSource);
165
166     protected override IntPtr GetRightPointer(ulong node) =>
167     ↳ new IntPtr(&Links[node].RightAsSource);
168
169     protected override ulong GetLeftValue(ulong node) =>
170     ↳ Links[node].LeftAsSource;
171
172     protected override ulong GetRightValue(ulong node) =>
173     ↳ Links[node].RightAsSource;
174
175     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 =
184     ↳ MathHelpers.PartialWrite(previousValue, size, 5,
185     ↳ -5);
186     var modified = (previousValue & 31) | ((size &
187     ↳ 134217727) << 5);
188     Links[node].SizeAsSource = modified;
189 }
190
191 protected override bool GetLeftIsChild(ulong node)
192 {
193     var previousValue = Links[node].SizeAsSource;
194     //return
195     ↳ (Integer)MathHelpers.PartialRead(previousValue, 4,
196     ↳ 1);
197     return (previousValue & 16) >> 4 == 1UL;
198 }
199
200 protected override void SetLeftIsChild(ulong node, bool
201     ↳ value)
202 {
203     var previousValue = Links[node].SizeAsSource;
204     //var modified =
205     ↳ MathHelpers.PartialWrite(previousValue,
206     ↳ (ulong)(Integer)value, 4, 1);
207     var modified = (previousValue & 4294967279) | ((value
208     ↳ ? 1UL : 0UL) << 4);
209     Links[node].SizeAsSource = modified;
210 }
211
212 protected override bool GetRightIsChild(ulong node)
213 {
214     var previousValue = Links[node].SizeAsSource;
215     //return
216     ↳ (Integer)MathHelpers.PartialRead(previousValue, 3,
217     ↳ 1);
218     return (previousValue & 8) >> 3 == 1UL;
219 }
220
221 protected override void SetRightIsChild(ulong node, bool
222     ↳ value)
223 {
224     var previousValue = Links[node].SizeAsSource;
225     //return
226     ↳ (Integer)MathHelpers.PartialRead(previousValue, 3,
227     ↳ 1);
228     return (previousValue & 8) >> 3 == 1UL;
229 }
230
231 protected override void SetLeftIsChild(ulong node, bool
232     ↳ value)
233 {
234     var previousValue = Links[node].SizeAsSource;
235     //var modified =
236     ↳ MathHelpers.PartialWrite(previousValue,
237     ↳ (ulong)(Integer)value, 4, 1);
238     var modified = (previousValue & 4294967279) | ((value
239     ↳ ? 1UL : 0UL) << 4);
240     Links[node].SizeAsSource = modified;
241 }
242
243 protected override void SetRightIsChild(ulong node, bool
244     ↳ value)
245 {
246     var previousValue = Links[node].SizeAsSource;
247     //var modified =
248     ↳ MathHelpers.PartialWrite(previousValue,
249     ↳ (ulong)(Integer)value, 3, 1);
250     var modified = (previousValue & 4294967287) | ((value
251     ↳ ? 1UL : 0UL) << 3);
252     Links[node].SizeAsSource = modified;
253 }
254
255 protected override sbyte GetBalance(ulong node)
256 {
257     var previousValue = Links[node].SizeAsSource;
258     //var value = MathHelpers.PartialRead(previousValue,
259     ↳ 0, 3);
260     var value = previousValue & 7;
261     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value
262     ↳ & 4) << 5) | value & 3 | 124 : value & 3);
263     return unpackedValue;
264 }
265
266 protected override void SetBalance(ulong node, sbyte value)
267 {
268     var previousValue = Links[node].SizeAsSource;
269     var packagedValue = (ulong)((byte)value >> 5) & 4 |
270     ↳ value & 3);
271     //var modified =
272     ↳ MathHelpers.PartialWrite(previousValue,
273     ↳ packagedValue, 0, 3);
274     var modified = (previousValue & 4294967288) |
275     ↳ (packagedValue & 7);
276     Links[node].SizeAsSource = modified;
277 }
278
279 protected override bool FirstIsToTheLeftOfSecond(ulong
280     ↳ first, ulong second)
281     => Links[first].Source < Links[second].Source ||
282     (Links[first].Source == Links[second].Source &&
283     ↳ Links[first].Target < Links[second].Target);
284
285 protected override bool FirstIsToTheRightOfSecond(ulong
286     ↳ first, ulong second)
287     => Links[first].Source > Links[second].Source ||
288     (Links[first].Source == Links[second].Source &&
289     ↳ Links[first].Target > Links[second].Target);
290
291 protected override ulong GetTreeRoot() =>
292     ↳ Header->FirstAsSource;
293
294 protected override ulong GetBasePartValue(ulong link) =>
295     ↳ Links[link].Source;
296
297 /// <summary>
298 /// Выполняет поиск и возвращает индекс связи с указанными
299     ↳ Source (началом) и Target (концом)
300 /// по дереву (индексу) связей, отсортированному по
301     ↳ Source, а затем по Target.
302 /// </summary>
303 /// <param name="source">Индекс связи, которая является
304     ↳ началом на искомой связи.</param>

```

```

211 //var modified =
212     ↳ MathHelpers.PartialWrite(previousValue,
213     ↳ (ulong)(Integer)value, 3, 1);
214 var modified = (previousValue & 4294967287) | ((value
215     ↳ ? 1UL : 0UL) << 3);
216 Links[node].SizeAsSource = modified;
217 }
218
219 protected override sbyte GetBalance(ulong node)
220 {
221     var previousValue = Links[node].SizeAsSource;
222     //var value = MathHelpers.PartialRead(previousValue,
223     ↳ 0, 3);
224     var value = previousValue & 7;
225     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value
226     ↳ & 4) << 5) | value & 3 | 124 : value & 3);
227     return unpackedValue;
228 }
229
230 protected override void SetBalance(ulong node, sbyte value)
231 {
232     var previousValue = Links[node].SizeAsSource;
233     var packagedValue = (ulong)((byte)value >> 5) & 4 |
234     ↳ value & 3);
235     //var modified =
236     ↳ MathHelpers.PartialWrite(previousValue,
237     ↳ packagedValue, 0, 3);
238     var modified = (previousValue & 4294967288) |
239     ↳ (packagedValue & 7);
240     Links[node].SizeAsSource = modified;
241 }
242
243 protected override bool FirstIsToTheLeftOfSecond(ulong
244     ↳ first, ulong second)
245     => Links[first].Source < Links[second].Source ||
246     (Links[first].Source == Links[second].Source &&
247     ↳ Links[first].Target < Links[second].Target);
248
249 protected override bool FirstIsToTheRightOfSecond(ulong
250     ↳ first, ulong second)
251     => Links[first].Source > Links[second].Source ||
252     (Links[first].Source == Links[second].Source &&
253     ↳ Links[first].Target > Links[second].Target);
254
255 protected override ulong GetTreeRoot() =>
256     ↳ Header->FirstAsSource;
257
258 protected override ulong GetBasePartValue(ulong link) =>
259     ↳ Links[link].Source;
260
261 /// <summary>
262 /// Выполняет поиск и возвращает индекс связи с указанными
263     ↳ Source (началом) и Target (концом)
264 /// по дереву (индексу) связей, отсортированному по
265     ↳ Source, а затем по Target.
266 /// </summary>
267 /// <param name="source">Индекс связи, которая является
268     ↳ началом на искомой связи.</param>

```

```

251 /// <param name="target">Индекс связи, которая является
252   ↳ концом на искомой связи.</param>
253 /// <returns>Индекс искомой связи.</returns>
254 public ulong Search(ulong source, ulong target)
255 {
256     var root = Header->FirstAsSource;
257     while (root != 0)
258     {
259         var rootSource = Links[root].Source;
260         var rootTarget = Links[root].Target;
261         if (FirstIsToTheLeftOfSecond(source, target,
262   ↳ rootSource, rootTarget)) // node.Key < root.Key
263         {
264             root = GetLeftOrDefault(root);
265         }
266         else if (FirstIsToTheRightOfSecond(source, target,
267   ↳ rootSource, rootTarget)) // node.Key > root.Key
268         {
269             root = GetRightOrDefault(root);
270         }
271         else // node.Key == root.Key
272         {
273             return root;
274         }
275     }
276     return 0;
277 }
278
279 [MethodImpl(MethodImplOptions.AggressiveInlining)]
280 private static bool FirstIsToTheLeftOfSecond(ulong
281   ↳ firstSource, ulong firstTarget, ulong secondSource,
282   ↳ ulong secondTarget)
283   => firstSource < secondSource || (firstSource ==
284   ↳ secondSource && firstTarget < secondTarget);
285
286 [MethodImpl(MethodImplOptions.AggressiveInlining)]
287 private static bool FirstIsToTheRightOfSecond(ulong
288   ↳ firstSource, ulong firstTarget, ulong secondSource,
289   ↳ ulong secondTarget)
290   => firstSource > secondSource || (firstSource ==
291   ↳ secondSource && firstTarget > secondTarget);
292
293 [MethodImpl(MethodImplOptions.AggressiveInlining)]
294 protected override void ClearNode(ulong node)
295 {
296     Links[node].LeftAsSource = OUL;
297     Links[node].RightAsSource = OUL;
298     Links[node].SizeAsSource = OUL;
299 }
300
301 [MethodImpl(MethodImplOptions.AggressiveInlining)]
302 protected override ulong GetZero() => OUL;
303
304 [MethodImpl(MethodImplOptions.AggressiveInlining)]
305 protected override ulong GetOne() => 1UL;
306
307 [MethodImpl(MethodImplOptions.AggressiveInlining)]
308 protected override ulong GetTwo() => 2UL;

```

```

301 [MethodImpl(MethodImplOptions.AggressiveInlining)]
302 protected override bool ValueEqualToZero(IntPtr pointer)
303   ↳ => *(ulong*)pointer.ToPointer() == OUL;
304
305 [MethodImpl(MethodImplOptions.AggressiveInlining)]
306 protected override bool EqualToZero(ulong value) => value
307   ↳ == OUL;
308
309 [MethodImpl(MethodImplOptions.AggressiveInlining)]
310 protected override bool IsEquals(ulong first, ulong
311   ↳ second) => first == second;
312
313 [MethodImpl(MethodImplOptions.AggressiveInlining)]
314 protected override bool GreaterThanZero(ulong value) =>
315   ↳ value > OUL;
316
317 [MethodImpl(MethodImplOptions.AggressiveInlining)]
318 protected override bool GreaterThan(ulong first, ulong
319   ↳ second) => first > second;
320
321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
322 protected override bool GreaterOrEqualThan(ulong first,
323   ↳ ulong second) => first >= second;
324
325 [MethodImpl(MethodImplOptions.AggressiveInlining)]
326 protected override bool GreaterOrEqualThanZero(ulong
327   ↳ value) => true; // value >= 0 is always true for ulong
328
329 [MethodImpl(MethodImplOptions.AggressiveInlining)]
330 protected override bool LessOrEqualThanZero(ulong value)
331   ↳ => value == 0; // value is always >= 0 for ulong
332
333 [MethodImpl(MethodImplOptions.AggressiveInlining)]
334 protected override bool LessOrEqualThan(ulong first, ulong
335   ↳ second) => first <= second;
336
337 [MethodImpl(MethodImplOptions.AggressiveInlining)]
338 protected override bool LessThanZero(ulong value) =>
339   ↳ false; // value < 0 is always false for ulong
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 protected override bool LessThan(ulong first, ulong
343   ↳ second) => first < second;
344
345 [MethodImpl(MethodImplOptions.AggressiveInlining)]
346 protected override ulong Increment(ulong value) => ++value;
347
348 [MethodImpl(MethodImplOptions.AggressiveInlining)]
349 protected override ulong Decrement(ulong value) => --value;
350
351 [MethodImpl(MethodImplOptions.AggressiveInlining)]
352 protected override ulong Add(ulong first, ulong second) =>
353   ↳ first + second;
354
355 [MethodImpl(MethodImplOptions.AggressiveInlining)]
356 protected override ulong Subtract(ulong first, ulong
357   ↳ second) => first - second;
358 }

```

```

347 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
348 {
349     public LinksTargetsTreeMethods(UInt64ResizableDirectMemory
        ↳ Links
        ↳ memory)
        : base(memory)
350     {
351     }
352
353     //protected override IntPtr GetLeft(ulong node) => new
        ↳ IntPtr(&Links[node].LeftAsTarget);
354
355     //protected override IntPtr GetRight(ulong node) => new
        ↳ IntPtr(&Links[node].RightAsTarget);
356
357     //protected override ulong GetSize(ulong node) =>
        ↳ Links[node].SizeAsTarget;
358
359     //protected override void SetLeft(ulong node, ulong left)
        ↳ => Links[node].LeftAsTarget = left;
360
361     //protected override void SetRight(ulong node, ulong
        ↳ right) => Links[node].RightAsTarget = right;
362
363     //protected override void SetSize(ulong node, ulong size)
        ↳ => Links[node].SizeAsTarget = size;
364
365     protected override IntPtr GetLeftPointer(ulong node) =>
        ↳ new IntPtr(&Links[node].LeftAsTarget);
366
367     protected override IntPtr GetRightPointer(ulong node) =>
        ↳ new IntPtr(&Links[node].RightAsTarget);
368
369     protected override ulong GetLeftValue(ulong node) =>
        ↳ Links[node].LeftAsTarget;
370
371     protected override ulong GetRightValue(ulong node) =>
        ↳ Links[node].RightAsTarget;
372
373     protected override ulong GetSize(ulong node)
374     {
375         var previousValue = Links[node].SizeAsTarget;
376         //return MathHelpers.PartialRead(previousValue, 5, -5);
377         return (previousValue & 4294967264) >> 5;
378     }
379
380     protected override void SetLeft(ulong node, ulong left) =>
        ↳ Links[node].LeftAsTarget = left;
381
382     protected override void SetRight(ulong node, ulong right)
        ↳ => Links[node].RightAsTarget = right;
383
384     protected override void SetSize(ulong node, ulong size)
385     {
386         var previousValue = Links[node].SizeAsTarget;
387         //var modified =
        ↳ MathHelpers.PartialWrite(previousValue, size, 5,
        ↳ -5);
388

```

```

        var modified = (previousValue & 31) | ((size &
        ↳ 134217727) << 5);
        Links[node].SizeAsTarget = modified;
    }

protected override bool GetLeftIsChild(ulong node)
{
    var previousValue = Links[node].SizeAsTarget;
    //return
        ↳ (Integer)MathHelpers.PartialRead(previousValue, 4,
        ↳ 1);
    return (previousValue & 16) >> 4 == 1UL;
    // TODO: Check if this is possible to use
    //var nodeSize = GetSize(node);
    //var left = GetLeftValue(node);
    //var leftSize = GetSizeOrZero(left);
    //return leftSize > 0 && nodeSize > leftSize;
}

protected override void SetLeftIsChild(ulong node, bool
    ↳ value)
{
    var previousValue = Links[node].SizeAsTarget;
    //var modified =
        ↳ MathHelpers.PartialWrite(previousValue,
        ↳ (ulong)(Integer)value, 4, 1);
    var modified = (previousValue & 4294967279) | ((value
        ↳ ? 1UL : 0UL) << 4);
    Links[node].SizeAsTarget = modified;
}

protected override bool GetRightIsChild(ulong node)
{
    var previousValue = Links[node].SizeAsTarget;
    //return
        ↳ (Integer)MathHelpers.PartialRead(previousValue, 3,
        ↳ 1);
    return (previousValue & 8) >> 3 == 1UL;
    // TODO: Check if this is possible to use
    //var nodeSize = GetSize(node);
    //var right = GetRightValue(node);
    //var rightSize = GetSizeOrZero(right);
    //return rightSize > 0 && nodeSize > rightSize;
}

protected override void SetRightIsChild(ulong node, bool
    ↳ value)
{
    var previousValue = Links[node].SizeAsTarget;
    //var modified =
        ↳ MathHelpers.PartialWrite(previousValue,
        ↳ (ulong)(Integer)value, 3, 1);
    var modified = (previousValue & 4294967287) | ((value
        ↳ ? 1UL : 0UL) << 3);
    Links[node].SizeAsTarget = modified;
}

protected override sbyte GetBalance(ulong node)

```

```

434 {
435     var previousValue = Links[node].SizeAsTarget;
436     //var value = MathHelpers.PartialRead(previousValue,
437     ↪ 0, 3);
438     var value = previousValue & 7;
439     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value
440     ↪ & 4) << 5) | value & 3 | 124 : value & 3);
441     return unpackedValue;
442 }
443 protected override void SetBalance(ulong node, sbyte value)
444 {
445     var previousValue = Links[node].SizeAsTarget;
446     var packagedValue = (ulong)((((byte)value >> 5) & 4) |
447     ↪ value & 3);
448     //var modified =
449     ↪ MathHelpers.PartialWrite(previousValue,
450     ↪ packagedValue, 0, 3);
451     var modified = (previousValue & 4294967288) |
452     ↪ (packagedValue & 7);
453     Links[node].SizeAsTarget = modified;
454 }
455 protected override bool FirstIsToTheLeftOfSecond(ulong
456 ↪ first, ulong second)
457 => Links[first].Target < Links[second].Target ||
458     (Links[first].Target == Links[second].Target &&
459     ↪ Links[first].Source < Links[second].Source);
460 protected override bool FirstIsToTheRightOfSecond(ulong
461 ↪ first, ulong second)
462 => Links[first].Target > Links[second].Target ||
463     (Links[first].Target == Links[second].Target &&
464     ↪ Links[first].Source > Links[second].Source);
465 protected override ulong GetTreeRoot() =>
466 ↪ Header->FirstAsTarget;
467 protected override ulong GetBasePartValue(ulong link) =>
468 ↪ Links[link].Target;
469 [MethodImpl(MethodImplOptions.AggressiveInlining)]
470 protected override void ClearNode(ulong node)
471 {
472     Links[node].LeftAsTarget = OUL;
473     Links[node].RightAsTarget = OUL;
474     Links[node].SizeAsTarget = OUL;
475 }
476 }
477 }
478 }

```

./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>

```

```

6 {
7     public BalancedVariantConverter(ILinks<TLink> links) :
8     ↪ base(links) { }
9
10 public override TLink Convert(ICollection<TLink> sequence)
11 {
12     var length = sequence.Count;
13     if (length < 1)
14     {
15         return default;
16     }
17     if (length == 1)
18     {
19         return sequence[0];
20     }
21     // Make copy of next layer
22     if (length > 2)
23     {
24         // TODO: Try to use stackalloc (which at the moment is
25         ↪ not working with generics) but will be possible
26         ↪ with Sigil
27         var halvedSequence = new TLink[(length / 2) + (length
28         ↪ % 2)];
29         HalveSequence(halvedSequence, sequence, length);
30         sequence = halvedSequence;
31         length = halvedSequence.Length;
32     }
33     // Keep creating layer after layer
34     while (length > 2)
35     {
36         HalveSequence(sequence, sequence, length);
37         length = (length / 2) + (length % 2);
38     }
39     return Links.GetOrCreate(sequence[0], sequence[1]);
40 }
41
42 private void HalveSequence(ICollection<TLink> destination,
43 ↪ ICollection<TLink> source, int length)
44 {
45     var loopedLength = length - (length % 2);
46     for (var i = 0; i < loopedLength; i += 2)
47     {
48         destination[i / 2] = Links.GetOrCreate(source[i],
49         ↪ source[i + 1]);
50     }
51     if (length > loopedLength)
52     {
53         destination[length / 2] = source[length - 1];
54     }
55 }

```

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

```

```

50     public CompressingConverter(ILinks<TLink> links,
51     ↪ IConverter<IList<TLink>, TLink> baseConverter,
52     ↪ LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
53     ↪ doInitialFrequenciesIncrement)
54     : this(links, baseConverter, doubletFrequenciesCache,
55     ↪ Integer<TLink>.One, doInitialFrequenciesIncrement)
56     {
57     }
58
59     public CompressingConverter(ILinks<TLink> links,
60     ↪ IConverter<IList<TLink>, TLink> baseConverter,
61     ↪ LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
62     ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
63     : base(links)
64     {
65         _baseConverter = baseConverter;
66         _doubletFrequenciesCache = doubletFrequenciesCache;
67         if (_comparer.Compare(minFrequencyToCompress,
68         ↪ Integer<TLink>.One) < 0)
69         {
70             minFrequencyToCompress = Integer<TLink>.One;
71         }
72         _minFrequencyToCompress = minFrequencyToCompress;
73         _doInitialFrequenciesIncrement =
74         ↪ doInitialFrequenciesIncrement;
75         ResetMaxDoublet();
76     }
77
78     public override TLink Convert(IList<TLink> source) =>
79     ↪ _baseConverter.Convert(Compress(source));
80
81     /// <remarks>
82     /// Original algorithm idea:
83     ↪ https://en.wikipedia.org/wiki/Byte_pair_encoding .
84     /// Faster version (doublets' frequencies dictionary is not
85     ↪ recreated).
86     /// </remarks>
87     private IList<TLink> Compress(IList<TLink> sequence)
88     {
89         if (sequence.IsNullOrEmpty())
90         {
91             return null;
92         }
93         if (sequence.Count == 1)
94         {
95             return sequence;
96         }
97         if (sequence.Count == 2)
98         {
99             return new[] { Links.GetOrCreate(sequence[0],
100             ↪ sequence[1]) };
101         }
102         // TODO: arraypool with min size (to improve cache
103         ↪ locality) or stackallow with Sigil
104         var copy = new HalfDoublet[sequence.Count];
105         Doublet<TLink> doublet = default;
106         for (var i = 1; i < sequence.Count; i++)
107         {
108             doublet.Source = sequence[i - 1];
109         }
110     }

```



```

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

```

```

140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186

```

```

        _maxDoubletData.Link =
        ↳ Links.GetOrCreate(maxDoubletSource,
        ↳ maxDoubletTarget);
    }
    var maxDoubletReplacementLink = _maxDoubletData.Link;
    oldLength--;
    var oldLengthMinusTwo = oldLength - 1;
    // Substitute all usages
    int w = 0, r = 0; // (r == read, w == write)
    for (; r < oldLength; r++)
    {
        if (_equalityComparer.Equals(copy[r].Element,
        ↳ maxDoubletSource) &&
        ↳ _equalityComparer.Equals(copy[r + 1].Element,
        ↳ maxDoubletTarget))
        {
            if (r > 0)
            {
                var previous = copy[w - 1].Element;
                copy[w -
                ↳ 1].DoubletData.DecrementFrequency();
                copy[w - 1].DoubletData = _doubletFrequenc
                ↳ iesCache.IncrementFrequency(previous,
                ↳ maxDoubletReplacementLink);
            }
            if (r < oldLengthMinusTwo)
            {
                var next = copy[r + 2].Element;
                copy[r +
                ↳ 1].DoubletData.DecrementFrequency();
                copy[w].DoubletData =
                ↳ _doubletFrequenciesCache.IncrementFreq
                ↳ uency(maxDoubletReplacementLink,
                ↳ next);
            }
            copy[w++].Element = maxDoubletReplacementLink;
            r++;
            newLength--;
        }
        else
        {
            copy[w++] = copy[r];
        }
    }
    if (w < newLength)
    {
        copy[w] = copy[r];
    }
    oldLength = newLength;
    ResetMaxDoublet();
    UpdateMaxDoublet(copy, newLength);
}
return newLength;
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
private void ResetMaxDoublet()
{
    _maxDoublet = new Doublet<TLink>();
}

```



```

48         GetPreviousLowerThanCurrentOrCurrent(p
        ↪     reviousLevel, currentLevel)
        ↪     :
49     i < 2 ?
50     GetNextLowerThanCurrentOrCurrent(curre
        ↪     ntLevel, levels[i + 1])
        ↪     :
51     GetGreatestNeighbourLowerThanCurrentOrC
        ↪     urrent(previousLevel,
        ↪     currentLevel, levels[i + 1]);
52     levels[w] = newLevel;
53     previousLevel = currentLevel;
54     w++;
55     levelRepeat = 0;
56     skipOnce = true;
57 }
58 else if (i == length - 1)
59 {
60     sequence[w] = sequence[i];
61     levels[w] = levels[i];
62     w++;
63 }
64 }
65 else
66 {
67     currentLevel = levels[i];
68     levelRepeat = 1;
69     if (skipOnce)
70     {
71         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
94 ↪     GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink
95 ↪     previous, TLink current, TLink next)
96 {
97     return _comparer.Compare(previous, next) > 0
98     ? _comparer.Compare(previous, current) < 0 ? previous
99     ↪     : current
100     : _comparer.Compare(next, current) < 0 ? next :
101     ↪     current;

```

```

98     }
99
100     private static TLink GetNextLowerThanCurrentOrCurrent(TLink
        ↪     current, TLink next) => _comparer.Compare(next, current) <
        ↪     0 ? next : current;
101
102     private static TLink
        ↪     GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink
        ↪     current) => _comparer.Compare(previous, current) < 0 ?
        ↪     previous : current;
103 }
104 }

```

./Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs

```

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

```

./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs

```

1 using Platform.Interfaces;
2
3 namespace Platform.Data.Doublets.Sequences.CreteriaMatchers

```

```

4  {
5      public class DefaultSequenceElementCriteriaMatcher<TLink> :
        ↳ LinksOperatorBase<TLink>, ICriteriaMatcher<TLink>
6      {
7          public DefaultSequenceElementCriteriaMatcher(ILinks<TLink>
            ↳ links) : base(links) { }
8          public bool IsMatched(TLink argument) =>
            ↳ Links.IsPartialPoint(argument);
9      }
10 }

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

```

```

./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>, ISequenceAppender<TLink>
9      {
10         private static readonly EqualityComparer<TLink>
            ↳ _equalityComparer = EqualityComparer<TLink>.Default;
11
12         private readonly IStack<TLink> _stack;
13         private readonly ISequenceHeightProvider<TLink>
            ↳ _heightProvider;
14

```

```

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

```

```

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

```

```

12     }
13 }

./Sequences/DuplicateSegmentsProvider.cs
1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using Platform.Interfaces;
5  using Platform.Collections;
6  using Platform.Collections.Lists;
7  using Platform.Collections.Segments;
8  using Platform.Collections.Segments.Walkers;
9  using Platform.Helpers;
10 using Platform.Helpers.Singletons;
11 using Platform.Numbers;
12 using Platform.Data.Sequences;
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     public class DuplicateSegmentsProvider<TLink> :
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>>>
23             ↳ _groups;
24         private BitString _visited;
25
26         private class ItemEqualityComparer :
27             ↳ IEqualityComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
28         {
29             private readonly IListEqualityComparer<TLink>
30                 ↳ _listComparer;
31             public ItemEqualityComparer() => _listComparer =
32                 ↳ Default<IListEqualityComparer<TLink>>.Instance;
33             public bool Equals(KeyValuePair<IList<TLink>,
34                               IList<TLink>> left, KeyValuePair<IList<TLink>,
35                               IList<TLink>> right) => _listComparer.Equals(left.Key,
36                               ↳ right.Key) && _listComparer.Equals(left.Value,
37                               ↳ right.Value);
38             public int GetHashCode(KeyValuePair<IList<TLink>,
39                                   IList<TLink>> pair) => HashHelpers.Generate(_listCompa
40                                   ↳ rer.GetHashCode(pair.Key),
41                                   ↳ _listComparer.GetHashCode(pair.Value));
42         }
43
44         private class ItemComparer :
45             ↳ IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
46         {
47             private readonly IListComparer<TLink> _listComparer;
48
49             public ItemComparer() => _listComparer =
50                 ↳ Default<IListComparer<TLink>>.Instance;
51
52             public int Compare(KeyValuePair<IList<TLink>,
53                               IList<TLink>> left, KeyValuePair<IList<TLink>,
54                               IList<TLink>> right)
55             {
56                 var intermediateResult =
57                     ↳ _listComparer.Compare(left.Key, right.Key);

```

```

40         if (intermediateResult == 0)
41         {
42             intermediateResult =
43                 ↳ _listComparer.Compare(left.Value, right.Value);
44         }
45         return intermediateResult;
46     }
47
48     public DuplicateSegmentsProvider(ILinks<TLink> links,
49         ↳ ISequences<TLink> 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.AddAndReturnT
70                     ↳ rue,
71                     ↳ linkIndex);
72                 if (sequenceElements.Count > 2)
73                 {
74                     WalkAll(sequenceElements);
75                 }
76                 return _links.Constants.Continue;
77             });
78             var resultList = _groups.ToList();
79             var comparer = Default<ItemComparer>.Instance;
80             resultList.Sort(comparer);
81
82             #if DEBUG
83             foreach (var item in resultList)
84             {
85                 PrintDuplicates(item);
86             }
87             return resultList;
88         }
89
90         #endif
91
92         protected override Segment<TLink> CreateSegment(IList<TLink>
93             ↳ elements, int offset, int length) => new
94             ↳ Segment<TLink>(elements, offset, length);
95
96         protected override void OnDuplicateFound(Segment<TLink>
97             ↳ segment)
98         {
99             var duplicates = CollectDuplicatesForSegment(segment);

```

```

92     if (duplicates.Count > 1)
93     {
94         _groups.Add(new KeyValuePair<IList<TLink>,
95                     ↳ IList<TLink>>(segment.ToArray(), duplicates));
96     }
97 }
98 private List<TLink> CollectDuplicatesForSegment(Segment<TLink>
99 ↳ segment)
100 {
101     var duplicates = new List<TLink>();
102     var readAsElement = new HashSet<TLink>();
103     _sequences.Each(sequence =>
104     {
105         duplicates.Add(sequence);
106         readAsElement.Add(sequence);
107         return true; // Continue
108     }, segment);
109     if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
110     {
111         return new List<TLink>();
112     }
113     foreach (var duplicate in duplicates)
114     {
115         var duplicateBitIndex =
116             ↳ (long)(Integer<TLink>)duplicate;
117         _visited.Set(duplicateBitIndex);
118     }
119     if (_sequences is Sequences sequencesExperiments)
120     {
121         var partiallyMatched =
122             ↳ sequencesExperiments.GetAllPartiallyMatchingSequen
123             ↳ ces4((HashSet<ulong>)(object)readAsElement,
124             ↳ (IList<ulong>)segment);
125         foreach (var partiallyMatchedSequence in
126             ↳ partiallyMatched)
127         {
128             TLink sequenceIndex =
129                 ↳ (Integer<TLink>)partiallyMatchedSequence;
130             duplicates.Add(sequenceIndex);
131         }
132     }
133     duplicates.Sort();
134     return duplicates;
135 }
136 private void PrintDuplicates(KeyValuePair<IList<TLink>,
137 ↳ IList<TLink>> duplicatesItem)
138 {
139     if (!(_links is ILinks<ulong> ulongLinks))
140     {
141         return;
142     }
143     var duplicatesKey = duplicatesItem.Key;
144     var keyString = UnicodeMap.FromLinksToString((IList<ulong>
145         ↳ )duplicatesKey);
146     Console.WriteLine($"> {keyString} ({string.Join(", ",
147         ↳ duplicatesKey)})");
148     var duplicatesList = duplicatesItem.Value;

```

```

140     for (int i = 0; i < duplicatesList.Count; i++)
141     {
142         ulong sequenceIndex =
143             ↳ (Integer<TLink>)duplicatesList[i];
144         var formattedSequenceStructure =
145             ↳ ulongLinks.FormatStructure(sequenceIndex, x =>
146             ↳ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
147             ↳ UnicodeMap.IsCharLink(link.Index) ?
148             ↳ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) :
149             ↳ sb.Append(link.Index));
150         Console.WriteLine(formattedSequenceStructure);
151         var sequenceString =
152             ↳ UnicodeMap.FromSequenceLinkToString(sequenceIndex,
153             ↳ ulongLinks);
154         Console.WriteLine(sequenceString);
155     }
156     Console.WriteLine();
157 }
158 }
159 }

```

./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
5 {
6     public class FrequenciesCacheBasedLinkFrequencyIncrementer<TLink>
7         ↳ : IIncrementer<IList<TLink>>
8     {
9         private readonly LinkFrequenciesCache<TLink> _cache;
10
11         public FrequenciesCacheBasedLinkFrequencyIncrementer(LinkFrequ
12             ↳ enciesCache<TLink> cache) => _cache =
13             ↳ cache;
14
15         /// <remarks>Sequence itseft is not changed, only frequency of
16             ↳ its doublets is 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
6         ↳ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>
7         ↳ : IConverter<Doublet<TLink>, TLink>
8     {
9         private readonly LinkFrequenciesCache<TLink> _cache;
10         public FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(
11             ↳ LinkFrequenciesCache<TLink> cache) => _cache =
12             ↳ cache;

```

```

9         public TLink Convert(Doublet<TLink> source) =>
10             ↪ _cache.GetFrequency(ref source).Frequency;
11     }
}

./Sequences/Frequencies/Cache/LinkFrequenciesCache.cs
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Interfaces;
5 using Platform.Numbers;
6
7 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
8 {
9     /// <remarks>
10    /// Can be used to operate with many CompressingConverters (to
11    ↪ keep global frequencies data between them).
12    /// TODO: Extract interface to implement frequencies storage
13    ↪ inside Links storage
14    /// </remarks>
15    public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
16    {
17        private static readonly EqualityComparer<TLink>
18        ↪ _equalityComparer = EqualityComparer<TLink>.Default;
19        private static readonly Comparer<TLink> _comparer =
20        ↪ Comparer<TLink>.Default;
21
22        private readonly Dictionary<Doublet<TLink>,
23        ↪ LinkFrequency<TLink>> _doubletsCache;
24        private readonly ICounter<TLink, TLink> _frequencyCounter;
25
26        public LinkFrequenciesCache(ILinks<TLink> links,
27        ↪ ICounter<TLink, TLink> frequencyCounter)
28        : base(links)
29        {
30            _doubletsCache = new Dictionary<Doublet<TLink>,
31            ↪ LinkFrequency<TLink>>(4096,
32            ↪ DoubletComparer<TLink>.Default);
33            _frequencyCounter = frequencyCounter;
34        }
35
36        [MethodImpl(MethodImplOptions.AggressiveInlining)]
37        public LinkFrequency<TLink> GetFrequency(TLink source, TLink
38        ↪ target)
39        {
40            var doublet = new Doublet<TLink>(source, target);
41            return GetFrequency(ref doublet);
42        }
43
44        [MethodImpl(MethodImplOptions.AggressiveInlining)]
45        public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink>
46        ↪ doublet)
47        {
48            _doubletsCache.TryGetValue(doublet, out
49            ↪ LinkFrequency<TLink> data);
50            return data;
51        }
52
53        public void IncrementFrequencies(IList<TLink> sequence)
54        {

```

```

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,
52     ↪ TLink target)
53     {
54         var doublet = new Doublet<TLink>(source, target);
55         return IncrementFrequency(ref doublet);
56     }
57
58     public void PrintFrequencies(IList<TLink> sequence)
59     {
60         for (var i = 1; i < sequence.Count; i++)
61         {
62             PrintFrequency(sequence[i - 1], sequence[i]);
63         }
64     }
65
66     public void PrintFrequency(TLink source, TLink target)
67     {
68         var number = GetFrequency(source, target).Frequency;
69         Console.WriteLine("{0},{1} - {2}", source, target,
70         ↪ number);
71     }
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public LinkFrequency<TLink> IncrementFrequency(ref
75     ↪ Doublet<TLink> doublet)
76     {
77         if (_doubletsCache.TryGetValue(doublet, out
78         ↪ LinkFrequency<TLink> data))
79         {
80             data.IncrementFrequency();
81         }
82         else
83         {
84             var link = Links.SearchOrDefault(doublet.Source,
85             ↪ doublet.Target);
86             data = new LinkFrequency<TLink>(Integer<TLink>.One,
87             ↪ link);
88             if (!_equalityComparer.Equals(link, default))
89             {
90                 data.Frequency =
91                 ↪ ArithmeticHelpers.Add(data.Frequency,
92                 ↪ _frequencyCounter.Count(link));
93             }
94             _doubletsCache.Add(doublet, data);
95         }
96         return data;
97     }
98
99     public void ValidateFrequencies()
100     {
101         foreach (var entry in _doubletsCache)
102         {

```

```

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
102        // ↳ `count` by 1?
103        if (((_comparer.Compare(frequency, count) > 0) &&
104            (_comparer.Compare(ArithmeticHelpers.Subtract(
105                frequency, count), Integer<TLink>.One) >
106                0))
107            || ((_comparer.Compare(count, frequency) > 0) &&
108                (_comparer.Compare(ArithmeticHelpers.Subtract(
109                    count, frequency), Integer<TLink>.One) >
110                    0)))
111        {
112            throw new
113                ↳ InvalidOperationException("Frequencies
114                ↳ validation failed.");
115        }
116    }
117    //else
118    //{
119    //    if (value.Frequency > 0)
120    //    {
121    //        var frequency = value.Frequency;
122    //        linkIndex = _createLink(entry.Key.Source,
123    //            ↳ entry.Key.Target);
124    //        var count = _countLinkFrequency(linkIndex);
125    //        if ((frequency > count && frequency - count
126    //            ↳ > 1) || (count > frequency && count - frequency >
127    //            ↳ 1))
128    //            throw new Exception("Frequencies
129    //            ↳ validation failed.");
130    //    }
131    //}
132    }
133    }
134    }
135    }

```

./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  }

```

```

16     public LinkFrequency() { }
17
18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     public void IncrementFrequency() => Frequency =
20         ↳ ArithmeticHelpers<TLink>.Increment(Frequency);
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     public void DecrementFrequency() => Frequency =
24         ↳ ArithmeticHelpers<TLink>.Decrement(Frequency);
25
26     public override string ToString() => $"{F: {Frequency}}, L:
27         ↳ {Link}";
28 }

```

./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs

```

1  using Platform.Interfaces;
2
3  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
4  {
5      public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
6          ↳ SequenceSymbolFrequencyOneOffCounter<TLink>
7      {
8          private readonly ICriteriaMatcher<TLink>
9              ↳ _markedSequenceMatcher;
10
11          public
12              ↳ MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink>
13              ↳ links, ICriteriaMatcher<TLink> markedSequenceMatcher,
14              ↳ TLink sequenceLink, TLink symbol)
15              : base(links, sequenceLink, symbol)
16              => _markedSequenceMatcher = markedSequenceMatcher;
17
18          public override TLink Count()
19          {
20              if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
21              {
22                  return default;
23              }
24              return base.Count();
25          }
26      }
27  }

```

./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> :
9          ↳ ICounter<TLink>
10      {
11          private static readonly EqualityComparer<TLink>
12              ↳ _equalityComparer = EqualityComparer<TLink>.Default;
13          private static readonly Comparer<TLink> _comparer =
14              ↳ Comparer<TLink>.Default;
15      }
16  }

```



```

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

```

./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs

```

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

```

```

16     public TLink Count(TLink argument) => new TotalMarkedSequenceS
17     ↪ ymbolFrequencyOneOffCounter<TLink>(_links,
18     ↪ _markedSequenceMatcher, argument).Count();
19 }
20
21 }

```

./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.

```

1 using Platform.Interfaces;
2 using Platform.Numbers;
3
4 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
5 {
6     public class
7     ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
8     ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
9     {
10         private readonly ICriteriaMatcher<TLink>
11         ↪ _markedSequenceMatcher;
12
13         public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<
14         ↪ TLink> links, ICriteriaMatcher<TLink>
15         ↪ markedSequenceMatcher, TLink symbol) : base(links, symbol)
16         => _markedSequenceMatcher = markedSequenceMatcher;
17
18         protected override void CountSequenceSymbolFrequency(TLink
19         ↪ link)
20         {
21             var symbolFrequencyCounter = new MarkedSequenceSymbolFreque
22             ↪ encyOneOffCounter<TLink>(_links,
23             ↪ _markedSequenceMatcher, link, _symbol);
24             _total = ArithmeticHelpers.Add(_total,
25             ↪ symbolFrequencyCounter.Count());
26         }
27     }
28 }

```

./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs

```

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

```

./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs

```

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

```

```

6  {
7  public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> :
    ↳ ICounter<TLink>
8  {
9      private static readonly EqualityComparer<TLink>
    ↳ _equalityComparer = EqualityComparer<TLink>.Default;
10     private static readonly Comparer<TLink> _comparer =
    ↳ Comparer<TLink>.Default;
11
12     protected readonly ILinks<TLink> _links;
13     protected readonly TLink _symbol;
14     protected readonly HashSet<TLink> _visits;
15     protected TLink _total;
16
17     public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink>
    ↳ links, TLink symbol)
    {
18         _links = links;
19         _symbol = symbol;
20         _visits = new HashSet<TLink>();
21         _total = default;
22     }
23
24     public TLink Count()
    {
25         if (_comparer.Compare(_total, default) > 0 ||
26             ↳ _visits.Count > 0)
27         {
28             return _total;
29         }
30         CountCore(_symbol);
31         return _total;
32     }
33
34     private void CountCore(TLink link)
    {
35         var any = _links.Constants.Any;
36         if (_equalityComparer.Equals(_links.Count(any, link),
37             ↳ default))
38         {
39             CountSequenceSymbolFrequency(link);
40         }
41         else
42         {
43             _links.Each(EachElementHandler, any, link);
44         }
45     }
46
47     protected virtual void CountSequenceSymbolFrequency(TLink link)
    {
48         var symbolFrequencyCounter = new
49             ↳ SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
50                 ↳ link, _symbol);
51         _total = ArithmeticHelpers.Add(_total,
52             ↳ symbolFrequencyCounter.Count());
53     }
54
55     private TLink EachElementHandler(IList<TLink> doublet)
    {
56         var constants = _links.Constants;

```

```

57         var doubletIndex = doublet[constants.IndexPart];
58         if (_visits.Add(doubletIndex))
59         {
60             CountCore(doubletIndex);
61         }
62         return constants.Continue;
63     }
64 }
65 }

./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>, ISequenceHeightProvider<TLink>
7      {
8          private static readonly EqualityComparer<TLink>
    ↳ _equalityComparer = EqualityComparer<TLink>.Default;
9
10         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             TLink height;
34             var heightValue = _propertyOperator.GetValue(sequence,
35                 ↳ _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(height ↪
        ↪     Value);
45     }
46     return height;
47 }
48 }
49 }

```

./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```

1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  namespace Platform.Data.Doublets.Sequences.HeightProviders
5  {
6      public class DefaultSequenceRightHeightProvider<TLink> :
        ↪     LinksOperatorBase<TLink>, ISequenceHeightProvider<TLink>
7      {
8          private readonly ICreteriaMatcher<TLink> _elementMatcher;
9
10         public DefaultSequenceRightHeightProvider(ILinks<TLink> links,
        ↪     ICreteriaMatcher<TLink> elementMatcher) : base(links) =>
        ↪     _elementMatcher = elementMatcher;
11
12         public TLink Get(TLink sequence)
13         {
14             var height = default(TLink);
15             var pairOrElement = sequence;
16             while (!_elementMatcher.IsMatched(pairOrElement))
17             {
18                 pairOrElement = Links.GetTarget(pairOrElement);
19                 height = ArithmeticHelpers.Increment(height);
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
        ↪     (подпоследовательностей),
25     /// через естественную группировку по unicode типам, все
        ↪     whitespace вместе, все символы вместе, все числа вместе и т.п.
26     /// + использовать ровно сбалансированный вариант, чтобы уменьшать
        ↪     вложенность (глубину графа)
27     ///
28     /// x*y - найти все связи между, в последовательностях любой
        ↪     формы, если не стоит ограничитель на то, что является
        ↪     последовательностью, а что нет,
29     /// то находятся любые структуры связей, которые содержат эти
        ↪     элементы именно в таком порядке.
30     ///
31     /// Рост последовательности слева и справа.
32     /// Поиск со звёздочкой.
33     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
34     /// так же проблема может быть решена при реализации дистанционных
        ↪     триггеров.
35     /// Нужны ли уникальные указатели вообще?
36     /// Что если обращение к информации будет происходить через
        ↪     содержимое всегда?
37     ///
38     /// Писать тесты.
39     ///
40     ///
41     /// Можно убрать зависимость от конкретной реализации Links,
42     /// на зависимость от абстрактного элемента, который может быть
        ↪     представлен несколькими способами.
43     ///
44     /// Можно ли как-то сделать один общий интерфейс
45     ///
46     ///
47     /// Блокчейн и/или гит для распределённой записи транзакций.
48     ///
49     /// </remarks>
50     public partial class Sequences : ISequences<ulong> //
        ↪     IList<string>, IList<ulong[]> (после завершения реализации
        ↪     Sequences)
51     {
52         private static readonly LinksCombinedConstants<bool, ulong,
        ↪     long> _constants = Default<LinksCombinedConstants<bool,
        ↪     ulong, long>>.Instance;
53

```

```

54  /// <summary>Возвращает значение ulong, обозначающее любое
    ↳ количество связей.</summary>
55  public const ulong ZeroOrMany = ulong.MaxValue;
56
57  public SequencesOptions<ulong> Options;
58  public readonly SynchronizedLinks<ulong> Links;
59  public readonly ISynchronization Sync;
60
61  public Sequences(SynchronizedLinks<ulong> links)
62      : this(links, new SequencesOptions<ulong>())
63  {
64  }
65
66  public Sequences(SynchronizedLinks<ulong> links,
    ↳ SequencesOptions<ulong> options)
67  {
68      Links = links;
69      Sync = links.SyncRoot;
70      Options = options;
71
72      Options.ValidateOptions();
73      Options.InitOptions(Links);
74  }
75
76  public bool IsSequence(ulong sequence)
77  {
78      return Sync.ExecuteReadOperation(() =>
79      {
80          if (Options.UseSequenceMarker)
81          {
82              return Options.MarkedSequenceMatcher.IsMatched(sequence);
83          }
84          return !Links.Unsync.IsPartialPoint(sequence);
85      });
86  }
87
88  [MethodImpl(MethodImplOptions.AggressiveInlining)]
89  private ulong GetSequenceByElements(ulong sequence)
90  {
91      if (Options.UseSequenceMarker)
92      {
93          return
94              ↳ Links.SearchOrDefault(Options.SequenceMarkerLink,
95              ↳ sequence);
96      }
97      return sequence;
98  }
99
100 private ulong GetSequenceElements(ulong sequence)
101 {
102     if (Options.UseSequenceMarker)
103     {
104         var linkContents = new
105             ↳ UInt64Link(Links.GetLink(sequence));
106         if (linkContents.Source == Options.SequenceMarkerLink)
107         {
108             return linkContents.Target;
109         }
110         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,
122             ↳ Options.SequenceMarkerLink, _constants.Any);
123     }
124     if (sequence.Length == 1) // Первая связь это адрес
125     {
126         if (sequence[0] == _constants.Null)
127         {
128             return 0;
129         }
130         if (sequence[0] == _constants.Any)
131         {
132             return Count();
133         }
134         if (Options.UseSequenceMarker)
135         {
136             return Links.Count(_constants.Any,
137                 ↳ Options.SequenceMarkerLink, sequence[0]);
138         }
139         return Links.Exists(sequence[0]) ? 1UL : 0;
140     }
141     throw new NotImplementedException();
142 }
143
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 =
159                 ↳ GetSequenceElements(restrictions[0]);
160             var sequenceLink =
161                 ↳ GetSequenceByElements(elementsLink);
162             if (sequenceLink != _constants.Null)
163             {
164                 return Links.Count(sequenceLink) +
165                     ↳ Links.Count(elementsLink) - 1;
166             }

```

```

162         return Links.Count(elementsLink);
163     }
164     return Links.Count(restrictions[0]);
165 }
166 throw new NotImplementedException();
167 }
168 #endregion
169 #region Create
170
171 public ulong Create(params ulong[] sequence)
172 {
173     return Sync.ExecuteWriteOperation(() =>
174     {
175         if (sequence.IsNullOrEmpty())
176         {
177             return _constants.Null;
178         }
179         Links.EnsureEachLinkExists(sequence);
180         return CreateCore(sequence);
181     });
182 }
183
184 private ulong CreateCore(params ulong[] sequence)
185 {
186     if (Options.UseIndex)
187     {
188         Options.Indexer.Index(sequence);
189     }
190     var sequenceRoot = default(ulong);
191     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExistence)
192     {
193         var matches = Each(sequence);
194         if (matches.Count > 0)
195         {
196             sequenceRoot = matches[0];
197         }
198     }
199     else if
200     {
201         (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
202     {
203         return CompactCore(sequence);
204     }
205     if (sequenceRoot == default)
206     {
207         sequenceRoot =
208             Options.LinksToSequenceConverter.Convert(sequence);
209     }
210     if (Options.UseSequenceMarker)
211     {
212         Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink,
213             k,
214             sequenceRoot);
215     }
216     return sequenceRoot; // Возвращаем корень
217     // последовательности (т.е. сами элементы)
218 }

```

```

215 #endregion
216
217 #region Each
218
219 public List<ulong> Each(params ulong[] sequence)
220 {
221     var results = new List<ulong>();
222     Each(results.AddAndReturnTrue, sequence);
223     return results;
224 }
225
226 public bool Each(Func<ulong, bool> handler, IList<ulong>
227     sequence)
228 {
229     return Sync.ExecuteReadOperation(() =>
230     {
231         if (sequence.IsNullOrEmpty())
232         {
233             return true;
234         }
235         Links.EnsureEachLinkIsAnyOrExists(sequence);
236         if (sequence.Count == 1)
237         {
238             var link = sequence[0];
239             if (link == _constants.Any)
240             {
241                 return Links.Unsync.Each(_constants.Any,
242                     sequence, handler);
243             }
244             return handler(link);
245         }
246         if (sequence.Count == 2)
247         {
248             return Links.Unsync.Each(sequence[0], sequence[1],
249                 handler);
250         }
251         if (Options.UseIndex &&
252             !Options.Indexer.CheckIndex(sequence))
253         {
254             return false;
255         }
256         return EachCore(handler, sequence);
257     });
258 }
259
260 private bool EachCore(Func<ulong, bool> handler, IList<ulong>
261     sequence)
262 {
263     var matcher = new Matcher(this, sequence, new
264         HashSet<LinkIndex>(), handler);
265     // TODO: Find out why matcher.HandleFullMatched executed
266     // twice for the same sequence Id.
267     Func<ulong, bool> innerHandler = Options.UseSequenceMarker
268         ? (Func<ulong, bool>)matcher.HandleFullMatchedSequence
269         : matcher.HandleFullMatched;
270     //if (sequence.Length >= 2)
271     if (!StepRight(innerHandler, sequence[0], sequence[1]))
272     {

```

```

265         return false;
266     }
267     var last = sequence.Count - 2;
268     for (var i = 1; i < last; i++)
269     {
270         if (!PartialStepRight(innerHandler, sequence[i],
271             ↳ sequence[i + 1]))
272         {
273             return false;
274         }
275     }
276     if (sequence.Count >= 3)
277     {
278         if (!StepLeft(innerHandler, sequence[sequence.Count -
279             ↳ 2], sequence[sequence.Count - 1]))
280         {
281             return false;
282         }
283     }
284     return true;
285 }
286
287 private bool PartialStepRight(Func<ulong, bool> handler, ulong
288     ↳ left, ulong right)
289 {
290     return Links.Unsync.Each(_constants.Any, left, doublet =>
291     {
292         if (!StepRight(handler, doublet, right))
293         {
294             return false;
295         }
296         if (left != doublet)
297         {
298             return PartialStepRight(handler, doublet, right);
299         }
300         return true;
301     });
302 }
303
304 private bool StepRight(Func<ulong, bool> handler, ulong left,
305     ↳ ulong right) => Links.Unsync.Each(left, _constants.Any,
306     ↳ rightStep => TryStepRightUp(handler, right, rightStep));
307
308 private bool TryStepRightUp(Func<ulong, bool> handler, ulong
309     ↳ right, ulong stepFrom)
310 {
311     var upStep = stepFrom;
312     var firstSource = Links.Unsync.GetTarget(upStep);
313     while (firstSource != right && firstSource != upStep)
314     {
315         upStep = firstSource;
316         firstSource = Links.Unsync.GetSource(upStep);
317     }
318     if (firstSource == right)
319     {
320         return handler(stepFrom);
321     }
322     return true;
323 }

```

```

319 private bool StepLeft(Func<ulong, bool> handler, ulong left,
320     ↳ ulong right) => Links.Unsync.Each(_constants.Any, right,
321     ↳ leftStep => TryStepLeftUp(handler, left, leftStep));
322
323 private bool TryStepLeftUp(Func<ulong, bool> handler, ulong
324     ↳ left, ulong stepFrom)
325 {
326     var upStep = stepFrom;
327     var firstTarget = Links.Unsync.GetSource(upStep);
328     while (firstTarget != left && firstTarget != upStep)
329     {
330         upStep = firstTarget;
331         firstTarget = Links.Unsync.GetTarget(upStep);
332     }
333     if (firstTarget == left)
334     {
335         return handler(stepFrom);
336     }
337     return true;
338 }
339
340 #endregion
341
342 #region Update
343
344 public ulong Update(ulong[] sequence, ulong[] newSequence)
345 {
346     if (sequence.IsNullOrEmpty() &&
347         ↳ newSequence.IsNullOrEmpty())
348     {
349         return _constants.Null;
350     }
351     if (sequence.IsNullOrEmpty())
352     {
353         return Create(newSequence);
354     }
355     if (newSequence.IsNullOrEmpty())
356     {
357         Delete(sequence);
358         return _constants.Null;
359     }
360     return Sync.ExecuteWriteOperation(() =>
361     {
362         Links.EnsureEachLinkIsAnyOrExists(sequence);
363         Links.EnsureEachLinkExists(newSequence);
364         return UpdateCore(sequence, newSequence);
365     });
366 }
367
368 private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
369 {
370     ulong bestVariant;
371     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew
372         ↳ && !sequence.EqualTo(newSequence))
373     {
374         bestVariant = CompactCore(newSequence);
375     }
376     else
377     {

```

```

373         bestVariant = CreateCore(newSequence);
374     }
375     // TODO: Check all options only ones before loop execution
376     // Возможно нужно две версии Each, возвращающий
377     // ↳ фактические последовательности и с маркером,
378     // ↳ или возможно даже возвращать и тот и тот вариант. С
379     // ↳ другой стороны все варианты можно получить имея только
380     // ↳ 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 private void UpdateOneCore(ulong sequence, ulong newSequence)
391 {
392     if (Options.UseGarbageCollection)
393     {
394         var sequenceElements = GetSequenceElements(sequence);
395         var sequenceElementsContents = new
396             ↳ UInt64Link(Links.GetLink(sequenceElements));
397         var sequenceLink =
398             ↳ GetSequenceByElements(sequenceElements);
399         var newSequenceElements =
400             ↳ GetSequenceElements(newSequence);
401         var newSequenceLink =
402             ↳ GetSequenceByElements(newSequenceElements);
403         if (Options.UseCascadeUpdate ||
404             ↳ CountReferences(sequence) == 0)
405         {
406             if (sequenceLink != _constants.Null)
407             {
408                 Links.Unsync.Merge(sequenceLink,
409                     ↳ newSequenceLink);
410             }
411             Links.Unsync.Merge(sequenceElements,
412                 ↳ newSequenceElements);
413         }
414         ClearGarbage(sequenceElementsContents.Source);
415         ClearGarbage(sequenceElementsContents.Target);
416     }
417     else
418     {
419         if (Options.UseSequenceMarker)
420         {
421             var sequenceElements =
422                 ↳ GetSequenceElements(sequence);
423             var sequenceLink =
424                 ↳ GetSequenceByElements(sequenceElements);
425             var newSequenceElements =
426                 ↳ GetSequenceElements(newSequence);
427             var newSequenceLink =
428                 ↳ GetSequenceByElements(newSequenceElements);
429             if (Options.UseCascadeUpdate ||
430                 ↳ CountReferences(sequence) == 0)

```

```

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

```

```

468     }
469     else
470     {
471         if (Options.UseSequenceMarker)
472         {
473             var sequenceElements = GetSequenceElements(link);
474             var sequenceLink =
475                 ↳ GetSequenceByElements(sequenceElements);
476             if (Options.UseCascadeDelete ||
477                 ↳ CountReferences(link) == 0)
478             {
479                 if (sequenceLink != _constants.Null)
480                 {
481                     Links.Unsync.Delete(sequenceLink);
482                 }
483                 Links.Unsync.Delete(link);
484             }
485         }
486         else
487         {
488             if (Options.UseCascadeDelete ||
489                 ↳ CountReferences(link) == 0)
490             {
491                 Links.Unsync.Delete(link);
492             }
493         }
494     }
495 }
496
497 #endregion
498
499 #region Compactification
500
501 /// <remarks>
502 /// bestVariant можно выбирать по максимальному числу
503   ↳ использований,
504 /// но балансированный позволяет гарантировать уникальность
505   ↳ (если есть возможность,
506   ↳ гарантировать его использование в других местах).
507 ///
508 /// Получается этот метод должен игнорировать
509   ↳ Options.EnforceSingleSequenceVersionOnWrite
510 /// </remarks>
511 public ulong Compact(params ulong[] sequence)
512 {
513     return Sync.ExecuteWriteOperation(() =>
514     {
515         if (sequence.IsNullOrEmpty())
516         {
517             return _constants.Null;
518         }
519         Links.EnsureEachLinkExists(sequence);
520         return CompactCore(sequence);
521     });
522 }
523
524 [MethodImpl(MethodImplOptions.AggressiveInlining)]
525 private ulong CompactCore(params ulong[] sequence) =>
526     ↳ UpdateCore(sequence, sequence);

```

```

520
521 #endregion
522
523 #region Garbage Collection
524
525 /// <remarks>
526 /// TODO: Добавить дополнительный обработчик / событие
527   ↳ CanBeDeleted которое можно определить извне или в
528   ↳ унаследованном классе
529 /// </remarks>
530 [MethodImpl(MethodImplOptions.AggressiveInlining)]
531 private bool IsGarbage(ulong link) => link !=
532     ↳ Options.SequenceMarkerLink &&
533     ↳ !Links.Unsync.IsPartialPoint(link) && Links.Count(link) ==
534     ↳ 0;
535
536 private void ClearGarbage(ulong link)
537 {
538     if (IsGarbage(link))
539     {
540         var contents = new UInt64Link(Links.GetLink(link));
541         Links.Unsync.Delete(link);
542         ClearGarbage(contents.Source);
543         ClearGarbage(contents.Target);
544     }
545 }
546
547 #endregion
548
549 #region Walkers
550
551 public bool EachPart(Func<ulong, bool> handler, ulong sequence)
552 {
553     return Sync.ExecuteReadOperation(() =>
554     {
555         var links = Links.Unsync;
556         var walker = new RightSequenceWalker<ulong>(links);
557         foreach (var part in walker.Walk(sequence))
558         {
559             if (!handler(links.GetIndex(part)))
560             {
561                 return false;
562             }
563         }
564         return true;
565     });
566 }
567
568 public class Matcher : RightSequenceWalker<ulong>
569 {
570     private readonly Sequences _sequences;
571     private readonly IList<LinkIndex> _patternSequence;
572     private readonly HashSet<LinkIndex> _linksInSequence;
573     private readonly HashSet<LinkIndex> _results;
574     private readonly Func<ulong, bool> _stopableHandler;
575     private readonly HashSet<ulong> _readAsElements;
576     private int _filterPosition;
577
578     public Matcher(Sequences sequences, IList<LinkIndex>
579         ↳ patternSequence, HashSet<LinkIndex> results,
580         ↳ Func<LinkIndex, bool> stopableHandler,
581         ↳ HashSet<LinkIndex> readAsElements = null)

```



```

574         : base(sequences.Links.Unsync)
575     {
576         _sequences = sequences;
577         _patternSequence = patternSequence;
578         _linksInSequence = new
            ↳ HashSet<LinkIndex>(patternSequence.Where(x => x !=
            ↳ _constants.Any && x != ZeroOrMany));
579         _results = results;
580         _stopableHandler = stopableHandler;
581         _readAsElements = readAsElements;
582     }
583
584     protected override bool IsElement(IList<ulong> link) =>
            ↳ base.IsElement(link) || (_readAsElements != null &&
            ↳ _readAsElements.Contains(Links.GetIndex(link))) ||
            ↳ _linksInSequence.Contains(Links.GetIndex(link));
585
586     public bool FullMatch(LinkIndex sequenceToMatch)
587     {
588         _filterPosition = 0;
589         foreach (var part in Walk(sequenceToMatch))
590         {
591             if (!FullMatchCore(Links.GetIndex(part)))
592             {
593                 break;
594             }
595         }
596         return _filterPosition == _patternSequence.Count;
597     }
598
599     private bool FullMatchCore(LinkIndex element)
600     {
601         if (_filterPosition == _patternSequence.Count)
602         {
603             _filterPosition = -2; // Длиннее чем нужно
604             return false;
605         }
606         if (_patternSequence[_filterPosition] != _constants.Any
607             && element != _patternSequence[_filterPosition])
608         {
609             _filterPosition = -1;
610             return false; // Начинается/Продолжается иначе
611         }
612         _filterPosition++;
613         return true;
614     }
615
616     public void AddFullMatchedToResults(ulong sequenceToMatch)
617     {
618         if (FullMatch(sequenceToMatch))
619         {
620             _results.Add(sequenceToMatch);
621         }
622     }
623
624     public bool HandleFullMatched(ulong sequenceToMatch)
625     {
626         if (FullMatch(sequenceToMatch) &&
            ↳ _results.Add(sequenceToMatch))
627         {
628             return _stopableHandler(sequenceToMatch);

```

```

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

```

```

683         return true; // Ищем дальше
684     }
685     public void AddPartialMatchedToResults(ulong
686         ↪ sequenceToMatch)
687     {
688         if (PartialMatch(sequenceToMatch))
689         {
690             _results.Add(sequenceToMatch);
691         }
692     }
693     public bool HandlePartialMatched(ulong sequenceToMatch)
694     {
695         if (PartialMatch(sequenceToMatch))
696         {
697             return _stopableHandler(sequenceToMatch);
698         }
699         return true;
700     }
701     public void
702     ↪ AddAllPartialMatchedToResults(IEnumerable<ulong>
703     ↪ sequencesToMatch)
704     {
705         foreach (var sequenceToMatch in sequencesToMatch)
706         {
707             if (PartialMatch(sequenceToMatch))
708             {
709                 _results.Add(sequenceToMatch);
710             }
711         }
712     }
713     public void AddAllPartialMatchedToResultsAndReadAsElements
714     ↪ (IEnumerable<ulong>
715     ↪ sequencesToMatch)
716     {
717         foreach (var sequenceToMatch in sequencesToMatch)
718         {
719             if (PartialMatch(sequenceToMatch))
720             {
721                 _readAsElements.Add(sequenceToMatch);
722                 _results.Add(sequenceToMatch);
723             }
724         }
725     }
726     }
727     #endregion
728 }
729 }

```

./Sequences/Sequences.Experiments.cs

```

1  using System;
2  using LinkIndex = System.UInt64;
3  using System.Collections.Generic;
4  using Stack = System.Collections.Generic.Stack<ulong>;
5  using System.Linq;
6  using System.Text;

```

```

7  using Platform.Collections;
8  using Platform.Numbers;
9  using Platform.Data.Exceptions;
10 using Platform.Data.Sequences;
11 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
12 using Platform.Data.Doublets.Sequences.Walkers;
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     partial class Sequences
17     {
18         #region Create All Variants (Not Practical)
19
20         /// <remarks>
21         /// Number of links that is needed to generate all variants for
22         /// sequence of length N corresponds to
23         ↪ https://oeis.org/A014143/list sequence.
24         /// </remarks>
25         public ulong[] CreateAllVariants2(ulong[] sequence)
26         {
27             return Sync.ExecuteWriteOperation(() =>
28             {
29                 if (sequence.IsNullOrEmpty())
30                 {
31                     return new ulong[0];
32                 }
33                 Links.EnsureEachLinkExists(sequence);
34                 if (sequence.Length == 1)
35                 {
36                     return sequence;
37                 }
38                 return CreateAllVariants2Core(sequence, 0,
39                     ↪ sequence.Length - 1);
40             });
41         }
42         private ulong[] CreateAllVariants2Core(ulong[] sequence, long
43         ↪ startAt, long stopAt)
44         {
45             #if DEBUG
46                 if ((stopAt - startAt) < 0)
47                 {
48                     throw new ArgumentOutOfRangeException(nameof(startAt),
49                         ↪ "startAt должен быть меньше или равен stopAt");
50                 }
51             #endif
52             if ((stopAt - startAt) == 0)
53             {
54                 return new[] { sequence[startAt] };
55             }
56             if ((stopAt - startAt) == 1)
57             {
58                 return new[] {
59                     ↪ Links.Unsync.CreateAndUpdate(sequence[startAt],
60                     ↪ sequence[stopAt]) };
61             }
62             var variants = new ulong[(ulong)MathHelpers.Catalan(stopAt
63                 ↪ - startAt)];
64             var last = 0;

```

```

59     for (var splitter = startAt; splitter < stopAt; splitter++)
60     {
61         var left = CreateAllVariants2Core(sequence, startAt,
        ↪ splitter);
62         var right = CreateAllVariants2Core(sequence, splitter
        ↪ + 1, stopAt);
63         for (var i = 0; i < left.Length; i++)
64         {
65             for (var j = 0; j < right.Length; j++)
66             {
67                 var variant =
        ↪ Links.Unsync.CreateAndUpdate(left[i],
        ↪ right[j]);
68                 if (variant == _constants.Null)
69                 {
70                     throw new
        ↪ NotImplementedException("Creation
        ↪ cancellation is not implemented.");
71                 }
72                 variants[last++] = variant;
73             }
74         }
75     }
76     return variants;
77 }
78
79 public List<ulong> CreateAllVariants1(params ulong[] sequence)
80 {
81     return Sync.ExecuteWriteOperation(() =>
82     {
83         if (sequence.IsNullOrEmpty())
84         {
85             return new List<ulong>();
86         }
87         Links.Unsync.EnsureEachLinkExists(sequence);
88         if (sequence.Length == 1)
89         {
90             return new List<ulong> { sequence[0] };
91         }
92         var results = new List<ulong>((int)MathHelpers.Catalan
        ↪ (sequence.Length));
93         return CreateAllVariants1Core(sequence, results);
94     });
95 }
96
97 private List<ulong> CreateAllVariants1Core(ulong[] sequence,
    ↪ List<ulong> results)
98 {
99     if (sequence.Length == 2)
100     {
101         var link = Links.Unsync.CreateAndUpdate(sequence[0],
        ↪ sequence[1]);
102         if (link == _constants.Null)
103         {
104             throw new NotImplementedException("Creation
        ↪ cancellation is not implemented.");
105         }
106         results.Add(link);

```

```

107     return results;
108 }
109 var innerSequenceLength = sequence.Length - 1;
110 var innerSequence = new ulong[innerSequenceLength];
111 for (var li = 0; li < innerSequenceLength; li++)
112 {
113     var link = Links.Unsync.CreateAndUpdate(sequence[li],
        ↪ sequence[li + 1]);
114     if (link == _constants.Null)
115     {
116         throw new NotImplementedException("Creation
        ↪ cancellation is not implemented.");
117     }
118     for (var isi = 0; isi < li; isi++)
119     {
120         innerSequence[isi] = sequence[isi];
121     }
122     innerSequence[li] = link;
123     for (var isi = li + 1; isi < innerSequenceLength;
        ↪ isi++)
124     {
125         innerSequence[isi] = sequence[isi + 1];
126     }
127     CreateAllVariants1Core(innerSequence, results);
128 }
129 return results;
130 }
131
132 #endregion
133
134 public HashSet<ulong> Each1(params ulong[] sequence)
135 {
136     var visitedLinks = new HashSet<ulong>(); // Заменить на
        ↪ bitstring
137     Each1(link =>
138     {
139         if (!visitedLinks.Contains(link))
140         {
141             visitedLinks.Add(link); // изучить почему
        ↪ случаются повторы
142         }
143         return true;
144     }, sequence);
145     return visitedLinks;
146 }
147
148 private void Each1(Func<ulong, bool> handler, params ulong[]
    ↪ sequence)
149 {
150     if (sequence.Length == 2)
151     {
152         Links.Unsync.Each(sequence[0], sequence[1], handler);
153     }
154     else
155     {
156         var innerSequenceLength = sequence.Length - 1;
157         for (var li = 0; li < innerSequenceLength; li++)
158         {
159             var left = sequence[li];
160             var right = sequence[li + 1];

```

```

161         if (left == 0 && right == 0)
162         {
163             continue;
164         }
165         var linkIndex = li;
166         ulong[] innerSequence = null;
167         Links.Unsync.Each(left, right, doublet =>
168         {
169             if (innerSequence == null)
170             {
171                 innerSequence = new
172                     ↳ ulong[innerSequenceLength];
173                 for (var isi = 0; isi < linkIndex; isi++)
174                 {
175                     innerSequence[isi] = sequence[isi];
176                 }
177                 for (var isi = linkIndex + 1; isi <
178                     ↳ innerSequenceLength; isi++)
179                 {
180                     innerSequence[isi] = sequence[isi + 1];
181                 }
182                 innerSequence[linkIndex] = doublet;
183                 Each1(handler, innerSequence);
184                 return _constants.Continue;
185             }
186         });
187     }
188 }
189 public HashSet<ulong> EachPart(params ulong[] sequence)
190 {
191     var visitedLinks = new HashSet<ulong>(); // Заменить на
192     ↳ bitstring
193     EachPartCore(link =>
194     {
195         if (!visitedLinks.Contains(link))
196         {
197             visitedLinks.Add(link); // изучить почему
198             ↳ случаются повторы
199         }
200         return true;
201     }, sequence);
202     return visitedLinks;
203 }
204 public void EachPart(Func<ulong, bool> handler, params ulong[]
205     ↳ sequence)
206 {
207     var visitedLinks = new HashSet<ulong>(); // Заменить на
208     ↳ bitstring
209     EachPartCore(link =>
210     {
211         if (!visitedLinks.Contains(link))
212         {
213             visitedLinks.Add(link); // изучить почему
214             ↳ случаются повторы
215             return handler(link);
216         }
217     }, sequence);
218 }
219 }
220 }
221 }
222 }
223 }
224 }
225 }
226 }
227 }
228 }
229 }
230 }
231 }
232 }
233 }
234 }
235 }
236 }
237 }
238 }
239 }
240 }
241 }
242 }
243 }
244 }
245 }
246 }
247 }
248 }
249 }
250 }
251 }
252 }
253 }
254 }
255 }
256 }
257 }
258 }
259 }
260 }
261 }
262 }
263 }
264 }
265 }
266 }

```

```

214     }, sequence);
215 }
216 }
217 }
218 }
219 }
220 }
221 }
222 }
223 }
224 }
225 }
226 }
227 }
228 }
229 }
230 }
231 }
232 }
233 }
234 }
235 }
236 }
237 }
238 }
239 }
240 }
241 }
242 }
243 }
244 }
245 }
246 }
247 }
248 }
249 }
250 }
251 }
252 }
253 }
254 }
255 }
256 }
257 }
258 }
259 }
260 }
261 }
262 }
263 }
264 }
265 }
266 }

```

```

private void EachPartCore(Func<ulong, bool> handler, params
↳ ulong[] sequence)
{
    if (sequence.IsNullOrEmpty())
    {
        return;
    }
    Links.EnsureEachLinkIsAnyOrExists(sequence);
    if (sequence.Length == 1)
    {
        var link = sequence[0];
        if (link > 0)
        {
            handler(link);
        }
        else
        {
            Links.Each(_constants.Any, _constants.Any,
                ↳ handler);
        }
    }
    else if (sequence.Length == 2)
    {
        // _links.Each(sequence[0], sequence[1], handler);
        // o_|      x_o ...
        // x_|      |___|
        Links.Each(sequence[1], _constants.Any, doublet =>
        {
            var match = Links.SearchOrDefault(sequence[0],
                ↳ doublet);
            if (match != _constants.Null)
            {
                handler(match);
            }
            return true;
        });
        // |_x      ... x_o
        // |_o      |___|
        Links.Each(_constants.Any, sequence[0], doublet =>
        {
            var match = Links.SearchOrDefault(doublet,
                ↳ sequence[1]);
            if (match != 0)
            {
                handler(match);
            }
            return true;
        });
        //      .x o_.
        //      |___|
        PartialStepRight(x => handler(x), sequence[0],
            ↳ sequence[1]);
    }
    else
    {

```

```

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 =
388             ↪ Links.SearchOrDefault(firstElement,
389             ↪ sequence[1]);
390         if (doublet != _constants.Null)
391         {
392             results.Add(doublet);
393         }
394         return results;
395     }
396     var linksInSequence = new HashSet<ulong>(sequence);
397     void handler(ulong result)
398     {
399         var filterPosition = 0;
400         StopableSequenceWalker.WalkRight(result,
401             ↪ Links.Unsync.GetSource,
402             ↪ Links.Unsync.GetTarget,
403             x => linksInSequence.Contains(x) ||
404             ↪ Links.Unsync.GetTarget(x) == x, x =>
405             {
406                 if (filterPosition == sequence.Length)
407                 {
408                     filterPosition = -2; // Длиннее
409                     ↪ чем нужно
410                     return false;
411                 }
412                 if (x != sequence[filterPosition])
413                 {
414                     filterPosition = -1;
415                     return false; // Начинается иначе
416                 }
417                 filterPosition++;
418             }
419             return true;
420         });
421         if (filterPosition == sequence.Length)
422         {
423             results.Add(result);
424         }
425     }
426     if (sequence.Length >= 2)
427     {
428         StepRight(handler, sequence[0], sequence[1]);
429     }
430     var last = sequence.Length - 2;
431     for (var i = 1; i < last; i++)
432     {
433         PartialStepRight(handler, sequence[i],
434             ↪ sequence[i + 1]);
435     }
436     if (sequence.Length >= 3)
437     {

```

```

431         StepLeft(handler, sequence[sequence.Length -
432             ↪ 2], sequence[sequence.Length - 1]);
433     }
434 }
435 }
436 }
437 }
438 }
439 }
440 }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }
449 }
450 }
451 }
452 }
453 }
454 }
455 }
456 }
457 }
458 }
459 }
460 }
461 }
462 }
463 }
464 }
465 }
466 }
467 }
468 }
469 }
470 }
471 }
472 }
473 }
474 }
475 }
476 }
477 }
478 }
479 }

StepLeft(handler, sequence[sequence.Length -
    ↪ 2], sequence[sequence.Length - 1]);
}
}
return results;
});
}

public HashSet<ulong> GetAllMatchingSequences1(params ulong[]
    ↪ sequence)
{
    return Sync.ExecuteReadOperation(() =>
    {
        var results = new HashSet<ulong>();
        if (sequence.Length > 0)
        {
            Links.EnsureEachLinkExists(sequence);
            var firstElement = sequence[0];
            if (sequence.Length == 1)
            {
                results.Add(firstElement);
                return results;
            }
            if (sequence.Length == 2)
            {
                var doublet =
                    ↪ Links.SearchOrDefault(firstElement,
                    ↪ sequence[1]);
                if (doublet != _constants.Null)
                {
                    results.Add(doublet);
                }
                return results;
            }
            var matcher = new Matcher(this, sequence, results,
                ↪ null);
            if (sequence.Length >= 2)
            {
                StepRight(matcher.AddFullMatchedToResults,
                    ↪ sequence[0], sequence[1]);
            }
            var last = sequence.Length - 2;
            for (var i = 1; i < last; i++)
            {
                PartialStepRight(matcher.AddFullMatchedToResul
                    ↪ ts, sequence[i], sequence[i +
                    ↪ 1]);
            }
            if (sequence.Length >= 3)
            {
                StepLeft(matcher.AddFullMatchedToResults,
                    ↪ sequence[sequence.Length - 2],
                    ↪ sequence[sequence.Length - 1]);
            }
        }
        return results;
    });
}

```

```

480 public const int MaxSequenceFormatSize = 200;
481
482 public string FormatSequence(LinkIndex sequenceLink, params
↳ LinkIndex[] knownElements) => FormatSequence(sequenceLink,
↳ (sb, x) => sb.Append(x), true, knownElements);
483
484 public string FormatSequence(LinkIndex sequenceLink,
↳ Action<StringBuilder, LinkIndex> elementToString, bool
↳ insertComma, params LinkIndex[] knownElements) =>
↳ Links.SyncRoot.ExecuteReadOperation(() =>
↳ FormatSequence(Links.Unsync, sequenceLink,
↳ elementToString, insertComma, knownElements));
485
486 private string FormatSequence(ILinks<LinkIndex> links,
↳ LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
↳ elementToString, bool insertComma, params LinkIndex[]
↳ knownElements)
487 {
488     var linksInSequence = new HashSet<ulong>(knownElements);
489     //var entered = new HashSet<ulong>();
490     var sb = new StringBuilder();
491     sb.Append('{');
492     if (links.Exists(sequenceLink))
493     {
494         StopableSequenceWalker.WalkRight(sequenceLink,
↳ links.GetSource, links.GetTarget,
↳ x => linksInSequence.Contains(x) ||
↳ links.IsPartialPoint(x), element => //
↳ entered.AddAndReturnVoid, x => { },
↳ entered.DoNotContains
495     {
496         if (insertComma && sb.Length > 1)
497         {
498             sb.Append(',');
499         }
500         //if (entered.Contains(element))
501         //{
502         //    sb.Append('{');
503         //    elementToString(sb, element);
504         //    sb.Append('');
505         //}
506         //else
507         elementToString(sb, element);
508         if (sb.Length < MaxSequenceFormatSize)
509         {
510             return true;
511         }
512         sb.Append(insertComma ? ", ..." : "...");
513         return false;
514     });
515     }
516     sb.Append('}');
517     return sb.ToString();
518 }
519
520 public string SafeFormatSequence(LinkIndex sequenceLink,
↳ params LinkIndex[] knownElements) =>
↳ SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x),
↳ true, knownElements);
521

```

```

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,
↳ x => linksInSequence.Contains(x) ||
↳ links.IsFullPoint(x),
↳ entered.AddAndReturnVoid, x => { },
↳ entered.DoNotContains, element =>
534     {
535         if (insertComma && sb.Length > 1)
536         {
537             sb.Append(',');
538         }
539         if (entered.Contains(element))
540         {
541             sb.Append('{');
542             elementToString(sb, element);
543             sb.Append('');
544         }
545         else
546         {
547             elementToString(sb, element);
548         }
549         if (sb.Length < MaxSequenceFormatSize)
550         {
551             return true;
552         }
553         sb.Append(insertComma ? ", ..." : "...");
554         return false;
555     });
556     }
557     sb.Append('}');
558     return sb.ToString();
559 }
560
561 public List<ulong> GetAllPartiallyMatchingSequences0(params
↳ ulong[] sequence)
562 {
563     return Sync.ExecuteReadOperation(() =>
564     {
565         if (sequence.Length > 0)
566

```

```

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,
580             ↪ Links.Unsync.GetSource,
581             ↪ Links.Unsync.GetTarget,
582             x => linksInSequence.Contains(x) ||
583             ↪ Links.Unsync.GetTarget(x) == x, x =>
584             {
585                 if (filterPosition == (sequence.Length
586                     ↪ - 1))
587                 {
588                     return false;
589                 }
590                 if (filterPosition >= 0)
591                 {
592                     if (x == sequence[filterPosition +
593                         ↪ 1])
594                     {
595                         filterPosition++;
596                     }
597                     else
598                     {
599                         return false;
600                     }
601                 }
602                 if (filterPosition < 0)
603                 {
604                     if (x == sequence[0])
605                     {
606                         filterPosition = 0;
607                     }
608                 }
609                 return true;
610             });
611         if (filterPosition == (sequence.Length - 1))
612         {
613             filteredResults.Add(result);
614         }
615     }
616     return filteredResults;
617 }
618 return new List<ulong>();
619 });
620
621 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params
622     ↪ ulong[] sequence)
623 {
624     return Sync.ExecuteReadOperation(() =>
625         ↪

```

```

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,
631             ↪ filteredResults, null);
632         matcher.AddAllPartialMatchedToResults(results);
633         return filteredResults;
634     }
635     return new HashSet<ulong>();
636 });
637
638 public bool GetAllPartiallyMatchingSequences2(Func<ulong,
639     ↪ bool> handler, params ulong[] sequence)
640 {
641     return Sync.ExecuteReadOperation(() =>
642     {
643         if (sequence.Length > 0)
644         {
645             Links.EnsureEachLinkExists(sequence);
646
647             var results = new HashSet<ulong>();
648             var filteredResults = new HashSet<ulong>();
649             var matcher = new Matcher(this, sequence,
650                 ↪ filteredResults, handler);
651             for (var i = 0; i < sequence.Length; i++)
652             {
653                 if (!AllUsagesCore1(sequence[i], results,
654                     ↪ matcher.HandlePartialMatched))
655                 {
656                     return false;
657                 }
658             }
659             return true;
660         }
661         return true;
662     });
663 }
664
665 //public HashSet<ulong>
666 //    GetAllPartiallyMatchingSequences3(params ulong[] sequence)
667 //{
668 //    return Sync.ExecuteReadOperation(() =>
669 //    {
670 //        if (sequence.Length > 0)
671 //        {
672 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
673
674 //            var firstResults = new HashSet<ulong>();
675 //            var lastResults = new HashSet<ulong>();
676
677 //            var first = sequence.First(x => x !=
678 //                ↪ 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
            ↳ e[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.AddAllPartialMatchedToResultsAndReadAsElem
        ↳ ents(results.OrderBy(x => x)); // OrderBy is a
        ↳ Hack
756     return filteredResults;
757 }
758 return new HashSet<ulong>();
759 });
760 }
761
762 // Does not work
763 public HashSet<ulong>
        ↳ GetAllPartiallyMatchingSequences5(HashSet<ulong>
        ↳ readAsElements, params ulong[] sequence)
764 {
765     var visited = new HashSet<ulong>();
766     var results = new HashSet<ulong>();
767     var matcher = new Matcher(this, sequence, visited, x => {
        ↳ results.Add(x); return true; }, readAsElements);
768     var last = sequence.Length - 1;
769     for (var i = 0; i < last; i++)
770     {

```

```

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

```

```

821         //for (var i = 0; i < last; i++)
822         //{
823             //var results = new List<ulong>();
824             //StepRight(results.Add, sequence[i],
825             ↪ sequence[i + 1]);
826             //PartialStepRight(results.Add,
827             ↪ sequence[i], sequence[i + 1]);
828             //if (results.Count > 0)
829             //matches.Add(results);
830             //else
831             //return results;
832             //if (matches.Count == 2)
833             //{
834                 //var merged = new List<ulong>();
835                 //for (var j = 0; j <
836                 ↪ matches[0].Count; j++)
837                 //for (var k = 0; k <
838                 ↪ matches[1].Count; k++)
839                 //CloseInnerConnections(merged.Add,
840                 ↪ matches[0][j], matches[1][k]);
841                 //if (merged.Count > 0)
842                 //matches = new List<List<ulong>>
843                 ↪ { merged };
844                 //else
845                 //return new List<ulong>();
846             //}
847             //if (matches.Count > 0)
848             //{
849                 //var usages = new HashSet<ulong>();
850                 //for (int i = 0; i < sequence.Length; i++)
851                 //{
852                     //AllUsagesCore(sequence[i], usages);
853                 //}
854                 //for (int i = 0; i < matches[0].Count;
855                 ↪ i++)
856                 //AllUsagesCore(matches[0][i],
857                 ↪ usages);
858                 //usages.UnionWith(matches[0]);
859                 //return usages.ToList();
860             //}
861             var firstLinkUsages = new HashSet<ulong>();
862             AllUsagesCore(sequence[0], firstLinkUsages);
863             firstLinkUsages.Add(sequence[0]);
864             //var previousMatchings =
865             ↪ firstLinkUsages.ToList(); //new List<ulong>()
866             ↪ { sequence[0] }; // or all sequences,
867             ↪ containing this element?
868             //return
869             ↪ GetAllPartiallyMatchingSequencesCore(sequence,
870             ↪ firstLinkUsages, 1).ToList();
871             var results = new HashSet<ulong>();
872             foreach (var match in
873             ↪ GetAllPartiallyMatchingSequencesCore(sequence,
874             ↪ firstLinkUsages, 1))
875             {
876                 AllUsagesCore(match, results);

```

```

863         }
864         return results.ToList();
865     }
866     return new List<ulong>();
867 });
868 }
869
870 /// <remarks>
871 /// TODO: Может потребоваться ограничение на уровень глубины
872 /// ↪ рекурсии
873 /// </remarks>
874 public HashSet<ulong> AllUsages(ulong link)
875 {
876     return Sync.ExecuteReadOperation(() =>
877     {
878         var usages = new HashSet<ulong>();
879         AllUsagesCore(link, usages);
880         return usages;
881     });
882 }
883
884 // При сборе всех использований (последовательностей) можно
885 // ↪ сохранять обратный путь к той связи с которой начинался
886 // ↪ поиск (STTTSSSTT),
887 // ↪ причём достаточно одного бита для хранения перехода влево
888 // ↪ или вправо
889 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
890 {
891     bool handler(ulong doublet)
892     {
893         if (usages.Add(doublet))
894         {
895             AllUsagesCore(doublet, usages);
896         }
897         return true;
898     }
899     Links.Unsync.Each(link, _constants.Any, handler);
900     Links.Unsync.Each(_constants.Any, link, handler);
901 }
902
903 public HashSet<ulong> AllBottomUsages(ulong link)
904 {
905     return Sync.ExecuteReadOperation(() =>
906     {
907         var visits = new HashSet<ulong>();
908         var usages = new HashSet<ulong>();
909         AllBottomUsagesCore(link, visits, usages);
910         return usages;
911     });
912 }
913
914 private void AllBottomUsagesCore(ulong link, HashSet<ulong>
915 ↪ visits, HashSet<ulong> usages)
916 {
917     bool handler(ulong doublet)
918     {
919         if (visits.Add(doublet))
920         {
921             AllBottomUsagesCore(doublet, visits, usages);
922         }
923     }
924     Links.Unsync.Each(link, _constants.Any, handler);
925     Links.Unsync.Each(_constants.Any, link, handler);
926 }
927
928 }
929
930 }
931
932 }
933
934 }
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971

```

```

918         return true;
919     }
920     if (Links.Unsync.Count(_constants.Any, link) == 0)
921     {
922         usages.Add(link);
923     }
924     else
925     {
926         Links.Unsync.Each(link, _constants.Any, handler);
927         Links.Unsync.Each(_constants.Any, link, handler);
928     }
929 }
930
931 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
932 {
933     if (Options.UseSequenceMarker)
934     {
935         var counter = new TotalMarkedSequenceSymbolFrequencyOn
936         ↪ eOffCounter<ulong>(Links,
937         ↪ Options.MarkedSequenceMatcher, symbol);
938         return counter.Count();
939     }
940     else
941     {
942         var counter = new TotalSequenceSymbolFrequencyOneOffCo
943         ↪ unter<ulong>(Links,
944         ↪ symbol);
945         return counter.Count();
946     }
947 }
948
949 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages,
950 ↪ Func<ulong, bool> outerHandler)
951 {
952     bool handler(ulong doublet)
953     {
954         if (usages.Add(doublet))
955         {
956             if (!outerHandler(doublet))
957             {
958                 return false;
959             }
960             if (!AllUsagesCore1(doublet, usages, outerHandler))
961             {
962                 return false;
963             }
964         }
965         return true;
966     }
967     return Links.Unsync.Each(link, _constants.Any, handler)
968     && Links.Unsync.Each(_constants.Any, link, handler);
969 }
970
971 public void CalculateAllUsages(ulong[] totals)
972 {
973     var calculator = new AllUsagesCalculator(Links, totals);
974     calculator.Calculate();
975 }
976
977 }
978
979 }
980
981 }
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

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,
984         ↪ ulong[] totals)
985     {
986         _links = links;
987         _totals = totals;
988     }
989
990     public void Calculate() => _links.Each(_constants.Any,
991         ↪ _constants.Any, CalculateCore);
992
993     private bool CalculateCore(ulong link)
994     {
995         if (_totals[link] == 0)
996         {
997             var total = 1UL;
998             _totals[link] = total;
999             var visitedChildren = new HashSet<ulong>();
1000             bool linkCalculator(ulong child)
1001             {
1002                 if (link != child &&
1003                     ↪ visitedChildren.Add(child))
1004                 {
1005                     total += _totals[child] == 0 ? 1 :
1006                         ↪ _totals[child];
1007                 }
1008                 return true;
1009             }
1010             _links.Unsync.Each(link, _constants.Any,
1011                 ↪ linkCalculator);
1012             _links.Unsync.Each(_constants.Any, link,
1013                 ↪ linkCalculator);
1014             _totals[link] = total;
1015         }
1016         return true;
1017     }
1018 }
1019
1020 private class AllUsagesCalculator2
1021 {
1022     private readonly SynchronizedLinks<ulong> _links;
1023     private readonly ulong[] _totals;
1024
1025     public AllUsagesCalculator2(SynchronizedLinks<ulong>
1026         ↪ links, ulong[] totals)
1027     {
1028         _links = links;
1029         _totals = totals;
1030     }

```

1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081

```

public void Calculate() => _links.Each(_constants.Any,
    ↪ _constants.Any, CalculateCore);

private bool IsElement(ulong link)
{
    // _linksInSequence.Contains(link) ||
    return _links.Unsync.GetTarget(link) == link ||
        ↪ _links.Unsync.GetSource(link) == link;
}

private bool CalculateCore(ulong link)
{
    // TODO: Проработать защиту от заикливания
    // Основано на SequenceWalker.WalkLeft
    Func<ulong, ulong> getSource = _links.Unsync.GetSource;
    Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
    Func<ulong, bool> isElement = IsElement;
    void visitLeaf(ulong parent)
    {
        if (link != parent)
        {
            _totals[parent]++;
        }
    }
    void visitNode(ulong parent)
    {
        if (link != parent)
        {
            _totals[parent]++;
        }
    }
    var stack = new Stack();
    var element = link;
    if (isElement(element))
    {
        visitLeaf(element);
    }
    else
    {
        while (true)
        {
            if (isElement(element))
            {
                if (stack.Count == 0)
                {
                    break;
                }
                element = stack.Pop();
                var source = getSource(element);
                var target = getTarget(element);
                // Обработка элемента
                if (isElement(target))
                {
                    visitLeaf(target);
                }
                if (isElement(source))
                {
                    visitLeaf(source);
                }
            }
        }
    }
}

```

```

1082         element = source;
1083     }
1084     else
1085     {
1086         stack.Push(element);
1087         visitNode(element);
1088         element = getTarget(element);
1089     }
1090 }
1091 }
1092 _totals[link]++;
1093 return true;
1094 }
1095 }
1096
1097 private class AllUsagesCollector
1098 {
1099     private readonly ILinks<ulong> _links;
1100     private readonly HashSet<ulong> _usages;
1101
1102     public AllUsagesCollector(ILinks<ulong> links,
1103         ↪ HashSet<ulong> usages)
1104     {
1105         _links = links;
1106         _usages = usages;
1107     }
1108
1109     public bool Collect(ulong link)
1110     {
1111         if (_usages.Add(link))
1112         {
1113             _links.Each(link, _constants.Any, Collect);
1114             _links.Each(_constants.Any, link, Collect);
1115         }
1116         return true;
1117     }
1118 }
1119
1120 private class AllUsagesCollector1
1121 {
1122     private readonly ILinks<ulong> _links;
1123     private readonly HashSet<ulong> _usages;
1124     private readonly ulong _continue;
1125
1126     public AllUsagesCollector1(ILinks<ulong> links,
1127         ↪ HashSet<ulong> usages)
1128     {
1129         _links = links;
1130         _usages = usages;
1131         _continue = _links.Constants.Continue;
1132     }
1133
1134     public ulong Collect(IList<ulong> link)
1135     {
1136         var linkIndex = _links.GetIndex(link);
1137         if (_usages.Add(linkIndex))
1138         {
1139             _links.Each(Collect, _constants.Any, linkIndex);
1140         }
1141         return _continue;
1142     }
1143 }

```

```

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
1149             ↪ usages)
1150         {
1151             _links = links;
1152             _usages = usages;
1153         }
1154
1155         public bool Collect(ulong link)
1156         {
1157             if (_usages.Add((long)link))
1158             {
1159                 _links.Each(link, _constants.Any, Collect);
1160                 _links.Each(_constants.Any, link, Collect);
1161             }
1162             return true;
1163         }
1164     }
1165
1166     private class AllUsagesIntersectingCollector
1167     {
1168         private readonly SynchronizedLinks<ulong> _links;
1169         private readonly HashSet<ulong> _intersectWith;
1170         private readonly HashSet<ulong> _usages;
1171         private readonly HashSet<ulong> _enter;
1172
1173         public AllUsagesIntersectingCollector(SynchronizedLinks<ul
1174             ↪ ong> links, HashSet<ulong> intersectWith,
1175             ↪ HashSet<ulong> usages)
1176         {
1177             _links = links;
1178             _intersectWith = intersectWith;
1179             _usages = usages;
1180             _enter = new HashSet<ulong>(); // защита от
1181             ↪ заикливания
1182         }
1183
1184         public bool Collect(ulong link)
1185         {
1186             if (_enter.Add(link))
1187             {
1188                 if (_intersectWith.Contains(link))
1189                 {
1190                     _usages.Add(link);
1191                 }
1192                 _links.Unsync.Each(link, _constants.Any, Collect);
1193                 _links.Unsync.Each(_constants.Any, link, Collect);
1194             }
1195             return true;
1196         }
1197     }
1198
1199     private void CloseInnerConnections(Action<ulong> handler,
1200         ↪ ulong left, ulong right)

```

```

1196 {
1197     TryStepLeftUp(handler, left, right);
1198     TryStepRightUp(handler, right, left);
1199 }
1200
1201 private void AllCloseConnections(Action<ulong> handler, ulong
    ↳ left, ulong right)
1202 {
1203     // Direct
1204     if (left == right)
1205     {
1206         handler(left);
1207     }
1208     var doublet = Links.Unsync.SearchOrDefault(left, right);
1209     if (doublet != _constants.Null)
1210     {
1211         handler(doublet);
1212     }
1213     // Inner
1214     CloseInnerConnections(handler, left, right);
1215     // Outer
1216     StepLeft(handler, left, right);
1217     StepRight(handler, left, right);
1218     PartialStepRight(handler, left, right);
1219     PartialStepLeft(handler, left, right);
1220 }
1221
1222 private HashSet<ulong>
    ↳ GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
    ↳ HashSet<ulong> previousMatchings, long startAt)
1223 {
1224     if (startAt >= sequence.Length) // ?
1225     {
1226         return previousMatchings;
1227     }
1228     var secondLinkUsages = new HashSet<ulong>();
1229     AllUsagesCore(sequence[startAt], secondLinkUsages);
1230     secondLinkUsages.Add(sequence[startAt]);
1231     var matchings = new HashSet<ulong>();
1232     //for (var i = 0; i < previousMatchings.Count; i++)
1233     foreach (var secondLinkUsage in secondLinkUsages)
1234     {
1235         foreach (var previousMatching in previousMatchings)
1236         {
1237             //AllCloseConnections(matchings.AddAndReturnVoid,
1238             ↳ previousMatching, secondLinkUsage);
1239             StepRight(matchings.AddAndReturnVoid,
1240             ↳ previousMatching, secondLinkUsage);
1241             TryStepRightUp(matchings.AddAndReturnVoid,
1242             ↳ secondLinkUsage, previousMatching);
1243             //PartialStepRight(matchings.AddAndReturnVoid,
1244             ↳ secondLinkUsage, sequence[startAt]); //
            ↳ почему-то эта ошибочная запись приводит к
            ↳ желаемым результатам.
1245             PartialStepRight(matchings.AddAndReturnVoid,
1246             ↳ previousMatching, secondLinkUsage);
1247         }
1248     }
1249     if (matchings.Count == 0)

```

```

1245 {
1246     return matchings;
1247 }
1248
1249 return GetAllPartiallyMatchingSequencesCore(sequence,
    ↳ matchings, startAt + 1); // ??
1250 }
1251
1252 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(Sy
    ↳ nchronizedLinks<ulong> links, params ulong[]
    ↳ sequence)
1253 {
1254     if (sequence == null)
1255     {
1256         return;
1257     }
1258     for (var i = 0; i < sequence.Length; i++)
1259     {
1260         if (sequence[i] != _constants.Any && sequence[i] !=
            ↳ ZeroOrMany && !links.Exists(sequence[i]))
1261         {
1262             throw new ArgumentLinkDoesNotExistsException<ulong>
            ↳ >(sequence[i],
            ↳ $"patternSequence[{i}]");
1263         }
1264     }
1265 }
1266
1267 // Pattern Matching -> Key To Triggers
1268 public HashSet<ulong> MatchPattern(params ulong[]
    ↳ patternSequence)
1269 {
1270     return Sync.ExecuteReadOperation(() =>
1271     {
1272         patternSequence = Simplify(patternSequence);
1273         if (patternSequence.Length > 0)
1274         {
1275             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links,
            ↳ patternSequence);
1276             var uniqueSequenceElements = new HashSet<ulong>();
1277             for (var i = 0; i < patternSequence.Length; i++)
1278             {
1279                 if (patternSequence[i] != _constants.Any &&
                    ↳ patternSequence[i] != 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);

```

```

1290         matcher.AddAllPatternMatchedToResults(results);
1291         return filteredResults;
1292     }
1293     return new HashSet<ulong>();
1294 });
1295 }
1296
1297 // Найти все возможные связи между указанным списком связей.
1298 // Находит связи между всеми указанными связями в любом
1299 //   порядке.
1300 // TODO: решить что делать с повторами (когда одни и те же
1301 //   элементы встречаются несколько раз в последовательности)
1302 public HashSet<ulong> GetAllConnections(params ulong[]
1303     linksToConnect)
1304 {
1305     return Sync.ExecuteReadOperation(() =>
1306     {
1307         var results = new HashSet<ulong>();
1308         if (linksToConnect.Length > 0)
1309         {
1310             Links.EnsureEachLinkExists(linksToConnect);
1311             AllUsagesCore(linksToConnect[0], results);
1312             for (var i = 1; i < linksToConnect.Length; i++)
1313             {
1314                 var next = new HashSet<ulong>();
1315                 AllUsagesCore(linksToConnect[i], next);
1316                 results.IntersectWith(next);
1317             }
1318             return results;
1319         }
1320     });
1321 }
1322
1323 public HashSet<ulong> GetAllConnections1(params ulong[]
1324     linksToConnect)
1325 {
1326     return Sync.ExecuteReadOperation(() =>
1327     {
1328         var results = new HashSet<ulong>();
1329         if (linksToConnect.Length > 0)
1330         {
1331             Links.EnsureEachLinkExists(linksToConnect);
1332             var collector1 = new
1333                 AllUsagesCollector(Links.Unsync, results);
1334             collector1.Collect(linksToConnect[0]);
1335             var next = new HashSet<ulong>();
1336             for (var i = 1; i < linksToConnect.Length; i++)
1337             {
1338                 var collector = new
1339                     AllUsagesCollector(Links.Unsync, next);
1340                 collector.Collect(linksToConnect[i]);
1341                 results.IntersectWith(next);
1342                 next.Clear();
1343             }
1344             return results;
1345         }
1346     });
1347 }

```

```

1343 public HashSet<ulong> GetAllConnections2(params ulong[]
1344     linksToConnect)
1345 {
1346     return Sync.ExecuteReadOperation(() =>
1347     {
1348         var results = new HashSet<ulong>();
1349         if (linksToConnect.Length > 0)
1350         {
1351             Links.EnsureEachLinkExists(linksToConnect);
1352             var collector1 = new AllUsagesCollector(Links,
1353                 results);
1354             collector1.Collect(linksToConnect[0]);
1355             //AllUsagesCore(linksToConnect[0], results);
1356             for (var i = 1; i < linksToConnect.Length; i++)
1357             {
1358                 var next = new HashSet<ulong>();
1359                 var collector = new
1360                     AllUsagesIntersectingCollector(Links,
1361                         results, next);
1362                 collector.Collect(linksToConnect[i]);
1363                 //AllUsagesCore(linksToConnect[i], next);
1364                 //results.IntersectWith(next);
1365                 results = next;
1366             }
1367             return results;
1368         }
1369     });
1370 }
1371
1372 public List<ulong> GetAllConnections3(params ulong[]
1373     linksToConnect)
1374 {
1375     return Sync.ExecuteReadOperation(() =>
1376     {
1377         var results = new BitString((long)Links.Unsync.Count()
1378             + 1); // new BitArray((int)_links.Total + 1);
1379         if (linksToConnect.Length > 0)
1380         {
1381             Links.EnsureEachLinkExists(linksToConnect);
1382             var collector1 = new
1383                 AllUsagesCollector2(Links.Unsync, results);
1384             collector1.Collect(linksToConnect[0]);
1385             for (var i = 1; i < linksToConnect.Length; i++)
1386             {
1387                 var next = new
1388                     BitString((long)Links.Unsync.Count() + 1);
1389                 //new BitArray((int)_links.Total + 1);
1390                 var collector = new
1391                     AllUsagesCollector2(Links.Unsync, next);
1392                 collector.Collect(linksToConnect[i]);
1393                 results = results.And(next);
1394             }
1395             return results.GetSetUInt64Indices();
1396         }
1397     });
1398 }
1399
1400 private static ulong[] Simplify(ulong[] sequence)
1401 {

```

```

1392 // Считаем новый размер последовательности
1393 long newLength = 0;
1394 var zeroOrManyStepped = false;
1395 for (var i = 0; i < sequence.Length; i++)
1396 {
1397     if (sequence[i] == ZeroOrMany)
1398     {
1399         if (zeroOrManyStepped)
1400         {
1401             continue;
1402         }
1403         zeroOrManyStepped = true;
1404     }
1405     else
1406     {
1407         //if (zeroOrManyStepped) Is it efficient?
1408         zeroOrManyStepped = false;
1409     }
1410     newLength++;
1411 }
1412 // Строим новую последовательность
1413 zeroOrManyStepped = false;
1414 var newSequence = new ulong[newLength];
1415 long j = 0;
1416 for (var i = 0; i < sequence.Length; i++)
1417 {
1418     //var current = zeroOrManyStepped;
1419     //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1420     //if (current && zeroOrManyStepped)
1421     //    continue;
1422     //var newZeroOrManyStepped = patternSequence[i] ==
1423     ↪ zeroOrMany;
1424     //if (zeroOrManyStepped && newZeroOrManyStepped)
1425     //    continue;
1426     //zeroOrManyStepped = newZeroOrManyStepped;
1427     if (sequence[i] == ZeroOrMany)
1428     {
1429         if (zeroOrManyStepped)
1430         {
1431             continue;
1432         }
1433         zeroOrManyStepped = true;
1434     }
1435     else
1436     {
1437         //if (zeroOrManyStepped) Is it efficient?
1438         zeroOrManyStepped = false;
1439     }
1440     newSequence[j++] = sequence[i];
1441 }
1442 return newSequence;
1443 }
1444 public static void TestSimplify()
1445 {
1446     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3,
1447     ↪ 4, ZeroOrMany, ZeroOrMany, ZeroOrMany, 4, ZeroOrMany,
1448     ↪ ZeroOrMany, ZeroOrMany };
1449     var simplifiedSequence = Simplify(sequence);
1450 }

```

```

1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500

```

```

public List<ulong> GetSimilarSequences() => new List<ulong>();

public void Prediction()
{
    //_links
    //sequences
}

#region From Triplets

//public static void DeleteSequence(Link sequence)
//{
//}

public List<ulong> CollectMatchingSequences(ulong[] links)
{
    if (links.Length == 1)
    {
        throw new Exception("Подпоследовательности с одним
        ↪ элементом не поддерживаются.");
    }
    var leftBound = 0;
    var rightBound = links.Length - 1;
    var left = links[leftBound++];
    var right = links[rightBound--];
    var results = new List<ulong>();
    CollectMatchingSequences(left, leftBound, links, right,
    ↪ rightBound, ref results);
    return results;
}

private void CollectMatchingSequences(ulong leftLink, int
    ↪ leftBound, ulong[] middleLinks, ulong rightLink, int
    ↪ rightBound, ref List<ulong> results)
{
    var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
    var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
    if (leftLinkTotalReferers <= rightLinkTotalReferers)
    {
        var nextLeftLink = middleLinks[leftBound];
        var elements = GetRightElements(leftLink,
        ↪ nextLeftLink);
        if (leftBound <= rightBound)
        {
            for (var i = elements.Length - 1; i >= 0; i--)
            {
                var element = elements[i];
                if (element != 0)
                {
                    CollectMatchingSequences(element,
                    ↪ leftBound + 1, middleLinks, rightLink,
                    ↪ rightBound, ref results);
                }
            }
        }
        else
        {
            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,
1514     ↪ nextRightLink);
1515     if (leftBound <= rightBound)
1516     {
1517         for (var i = elements.Length - 1; i >= 0; i--)
1518         {
1519             var element = elements[i];
1520             if (element != 0)
1521             {
1522                 CollectMatchingSequences(leftLink,
1523                 ↪ leftBound, middleLinks, elements[i],
1524                 ↪ rightBound - 1, ref results);
1525             }
1526         }
1527     }
1528     else
1529     {
1530         for (var i = elements.Length - 1; i >= 0; i--)
1531         {
1532             var element = elements[i];
1533             if (element != 0)
1534             {
1535                 results.Add(element);
1536             }
1537         }
1538     }
1539 }
1540 }
1541 }
1542 }
1543 }
1544 }
1545 }
1546 }
1547 }
1548 }
1549 }
1550 }
1551 }
1552 }
1553 }
1554 }
1555 }
1556 }
1557 }
1558 }
1559 }
1560 }
1561 }
1562 }
1563 }
1564 }
1565 }
1566 }
1567 }
1568 }
1569 }
1570 }
1571 }
1572 }
1573 }
1574 }
1575 }
1576 }
1577 }
1578 }
1579 }
1580 }
1581 }
1582 }
1583 }
1584 }
1585 }
1586 }
1587 }
1588 }
1589 }
1590 }
1591 }
1592 }
1593 }
1594 }
1595 }
1596 }
1597 }
1598 }
1599 }
1600 }
1601 }
1602 }
1603 }
1604 }
1605 }
1606 }
1607 }
1608 }
1609 }
1610 }

```

```

1555     {
1556         result[4] = startLink;
1557     }
1558     return result;
1559 }
1560 }
1561 }
1562 }
1563 }
1564 }
1565 }
1566 }
1567 }
1568 }
1569 }
1570 }
1571 }
1572 }
1573 }
1574 }
1575 }
1576 }
1577 }
1578 }
1579 }
1580 }
1581 }
1582 }
1583 }
1584 }
1585 }
1586 }
1587 }
1588 }
1589 }
1590 }
1591 }
1592 }
1593 }
1594 }
1595 }
1596 }
1597 }
1598 }
1599 }
1600 }
1601 }
1602 }
1603 }
1604 }
1605 }
1606 }
1607 }
1608 }
1609 }
1610 }

```

```

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

```

```

1671     private int _patternPosition;
1672     private long _sequencePosition;
1673
1674     #endregion
1675
1676     public PatternMatcher(Sequences sequences, LinkIndex[]
1677     ↪     patternSequence, HashSet<LinkIndex> results)
1678     : base(sequences.Links.Unsync)
1679     {
1680         _sequences = sequences;
1681         _patternSequence = patternSequence;
1682         _linksInSequence = new
1683         ↪     HashSet<LinkIndex>(patternSequence.Where(x => x !=
1684         ↪     _constants.Any && x != ZeroOrMany));
1685         _results = results;
1686         _pattern = CreateDetailedPattern();
1687
1688     }
1689
1690     protected override bool IsElement(IList<ulong> link) =>
1691     ↪     _linksInSequence.Contains(Links.GetIndex(link)) ||
1692     ↪     base.IsElement(link);
1693
1694     public bool PatternMatch(LinkIndex sequenceToMatch)
1695     {
1696         _patternPosition = 0;
1697         _sequencePosition = 0;
1698         foreach (var part in Walk(sequenceToMatch))
1699         {
1700             if (!PatternMatchCore(Links.GetIndex(part)))
1701             {
1702                 break;
1703             }
1704         }
1705         return _patternPosition == _pattern.Count ||
1706         ↪     (_patternPosition == _pattern.Count - 1 &&
1707         ↪     _pattern[_patternPosition].Start == 0);
1708     }
1709
1710     private List<PatternBlock> CreateDetailedPattern()
1711     {
1712         var pattern = new List<PatternBlock>();
1713         var patternBlock = new PatternBlock();
1714         for (var i = 0; i < _patternSequence.Length; i++)
1715         {
1716             if (patternBlock.Type ==
1717             ↪     PatternBlockType.Undefined)
1718             {
1719                 if (_patternSequence[i] == _constants.Any)
1720                 {
1721                     patternBlock.Type = PatternBlockType.Gap;
1722                     patternBlock.Start = 1;
1723                     patternBlock.Stop = 1;
1724                 }
1725                 else if (_patternSequence[i] == ZeroOrMany)
1726                 {
1727                     patternBlock.Type = PatternBlockType.Gap;
1728                     patternBlock.Start = 0;
1729                     patternBlock.Stop = long.MaxValue;
1730                 }
1731             }

```

```

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 else // patternBlock.Type == PatternBlockType.Gap
1757 {
1758     if (_patternSequence[i] == _constants.Any)
1759     {
1760         patternBlock.Start++;
1761         if (patternBlock.Stop < patternBlock.Start)
1762         {
1763             patternBlock.Stop = patternBlock.Start;
1764         }
1765     }
1766     else if (_patternSequence[i] == ZeroOrMany)
1767     {
1768         patternBlock.Stop = long.MaxValue;
1769     }
1770     else
1771     {
1772         pattern.Add(patternBlock);
1773         patternBlock = new PatternBlock
1774         {
1775             Type = PatternBlockType.Elements,
1776             Start = i,
1777             Stop = i
1778         };
1779     }
1780 }

```

```

1781 }
1782 if (patternBlock.Type != PatternBlockType.Undefined)
1783 {
1784     pattern.Add(patternBlock);
1785 }
1786 return pattern;
1787 }
1788
1789 /** match: search for regexp anywhere in text */
1790 //int match(char* regexp, char* text)
1791 //{
1792 //    do
1793 //    {
1794 //        while (*text++ != '\0');
1795 //        return 0;
1796 //    }
1797
1798 /** matchhere: search for regexp at beginning of text */
1799 //int matchhere(char* regexp, char* text)
1800 //{
1801 //    if (regexp[0] == '\0')
1802 //        return 1;
1803 //    if (regexp[1] == '*')
1804 //        return matchstar(regexp[0], regexp + 2, text);
1805 //    if (regexp[0] == '$' && regexp[1] == '\0')
1806 //        return *text == '\0';
1807 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0]
1808 //        ↳ == *text))
1809 //        return matchhere(regexp + 1, text + 1);
1810 //    return 0;
1811 //}
1812
1813 /** matchstar: search for c*regexp at beginning of text */
1814 //int matchstar(int c, char* regexp, char* text)
1815 //{
1816 //    do
1817 //    {
1818 //        /* a * matches zero or more instances */
1819 //        if (matchhere(regexp, text))
1820 //            return 1;
1821 //    } while (*text != '\0' && (*text++ == c || c ==
1822 //        ↳ '.'));
1823 //    return 0;
1824 //}
1825
1826 //private void GetNextPatternElement(out LinkIndex
1827 //    ↳ element, out long minimumGap, out long maximumGap)
1828 //{
1829 //    minimumGap = 0;
1830 //    maximumGap = 0;
1831 //    element = 0;
1832 //    for (; _patternPosition < _patternSequence.Length;
1833 //        ↳ _patternPosition++)
1834 //    {
1835 //        if (_patternSequence[_patternPosition] ==
1836 //            ↳ Doublets.Links.Null)
1837 //            minimumGap++;
1838 //        else if (_patternSequence[_patternPosition] ==
1839 //            ↳ ZeroOrMany)

```

```

1833 //         maximumGap = long.MaxValue;
1834 //     else
1835 //         break;
1836 // }
1837
1838 // if (maximumGap < minimumGap)
1839 //     maximumGap = minimumGap;
1840 //}
1841
1842 private bool PatternMatchCore(LinkIndex element)
1843 {
1844     if (_patternPosition >= _pattern.Count)
1845     {
1846         _patternPosition = -2;
1847         return false;
1848     }
1849     var currentPatternBlock = _pattern[_patternPosition];
1850     if (currentPatternBlock.Type == PatternBlockType.Gap)
1851     {
1852         //var currentMatchingBlockLength =
1853         //    ↳ (_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         else
1868         {
1869             if (_sequencePosition >
1870                 ↳ currentPatternBlock.Stop)
1871             {
1872                 return false; // Соответствие невозможно
1873             }
1874             var nextPatternBlock =
1875                 ↳ _pattern[_patternPosition + 1];
1876             if (_patternSequence[nextPatternBlock.Start]
1877                 ↳ == element)
1878             {
1879                 if (nextPatternBlock.Start <
1880                     ↳ nextPatternBlock.Stop)
1881                 {
1882                     _patternPosition++;
1883                     _sequencePosition = 1;
1884                 }
1885                 else
1886                 {
1887                     _patternPosition += 2;
1888                     _sequencePosition = 0;
1889                 }
1890             }
1891         }
1892     }
1893 }

```

```

1887 else // currentPatternBlock.Type ==
1888     ↳ PatternBlockType.Elements
1889 {
1890     var patternElementPosition =
1891         ↳ currentPatternBlock.Start + _sequencePosition;
1892     if (_patternSequence[patternElementPosition] !=
1893         ↳ element)
1894     {
1895         return false; // Соответствие невозможно
1896     }
1897     if (patternElementPosition ==
1898         ↳ currentPatternBlock.Stop)
1899     {
1900         _patternPosition++;
1901         _sequencePosition = 0;
1902     }
1903     else
1904     {
1905         _sequencePosition++;
1906     }
1907 }
1908 return true;
1909 //if (_patternSequence[_patternPosition] != element)
1910 //    return false;
1911 //else
1912 //{
1913 //    _sequencePosition++;
1914 //    _patternPosition++;
1915 //    return true;
1916 //}
1917 //if (_filterPosition == _patternSequence.Length)
1918 //{
1919 //    _filterPosition = -2; // Длиннее чем нужно
1920 //    return false;
1921 //}
1922 //if (element != _patternSequence[_filterPosition])
1923 //{
1924 //    _filterPosition = -1;
1925 //    return false; // Начинается иначе
1926 //}
1927 //if (_filterPosition >= 0)
1928 //{
1929 //    if (element == _patternSequence[_filterPosition
1930 //        ↳ + 1])
1931 //        _filterPosition++;
1932 //    else
1933 //        return false;
1934 //}
1935 //if (_filterPosition < 0)
1936 //{
1937 //    if (element == _patternSequence[0])
1938 //        _filterPosition = 0;
1939 //}

```

```

1940     public void
1941     ↪ AddAllPatternMatchedToResults(IEnumerable<ulong>
1942     ↪ sequencesToMatch)
1943     {
1944         foreach (var sequenceToMatch in sequencesToMatch)
1945         {
1946             if (PatternMatch(sequenceToMatch))
1947             {
1948                 _results.Add(sequenceToMatch);
1949             }
1950         }
1951     }
1952 }
1953 #endregion
1954 }
1955 }

```

./Sequences/Sequences.Experiments.ReadSequence.cs

```

1  // #define USEARRAYPOOL
2  using System;
3  using System.Runtime.CompilerServices;
4  #if USEARRAYPOOL
5  using Platform.Collections;
6  #endif
7
8  namespace Platform.Data.Doublets.Sequences
9  {
10     partial class Sequences
11     {
12         public ulong[] ReadSequenceCore(ulong sequence, Func<ulong,
13         ↪ bool> isElement)
14         {
15             var links = Links.Unsync;
16             var length = 1;
17             var array = new ulong[length];
18             array[0] = sequence;
19
20             if (isElement(sequence))
21             {
22                 return array;
23             }
24
25             bool hasElements;
26             do
27             {
28                 length *= 2;
29                 #if USEARRAYPOOL
30                 var nextArray = ArrayPool.Allocate<ulong>(length);
31                 #else
32                 var nextArray = new ulong[length];
33                 #endif
34                 hasElements = false;
35                 for (var i = 0; i < array.Length; i++)
36                 {
37                     var candidate = array[i];
38                     if (candidate == 0)
39                     {
40                         continue;

```

```

41         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) &&
56                 ↪ isElement(linkTarget));
57             }
58         }
59     }
60     #if USEARRAYPOOL
61     if (array.Length > 1)
62     {
63         ArrayPool.Free(array);
64     }
65     #endif
66     array = nextArray;
67     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
81 ↪ filledElementsCount)
82 {
83     var finalArray = new ulong[filledElementsCount];
84     for (int i = 0, j = 0; i < array.Length; i++)
85     {
86         if (array[i] > 0)
87         {
88             finalArray[j] = array[i];
89             j++;
90         }
91     }
92     #if USEARRAYPOOL
93     ArrayPool.Free(array);
94     #endif
95     return finalArray;
96 }
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 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>
9              ↪ sequences, IList<TLink[]> groupedSequence)
10         {
11             var finalSequence = new TLink[groupedSequence.Count];
12             for (var i = 0; i < finalSequence.Length; i++)
13             {
14                 var part = groupedSequence[i];
15                 finalSequence[i] = part.Length == 1 ? part[0] :
16                     ↪ sequences.Create(part);
17             }
18             return sequences.Create(finalSequence);
19         }
20     }
21 }

```

./Sequences/SequencesIndexer.cs

```

1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Sequences
4  {
5      public class SequencesIndexer<TLink>
6      {
7          private static readonly EqualityComparer<TLink>
8              ↪ _equalityComparer = 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
21         ↪ значение,
22         /// определяющие была ли запрошенная последовательность
23         ↪ проиндексирована ранее.

```

```

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

```

```

70     }
71     return indexed;
72 }
73
74 public bool CheckIndex(IList<TLink> sequence)
75 {
76     var indexed = true;
77     var i = sequence.Count;
78     while (--i >= 1 && (indexed = !_equalityComparer.Equals(_l
79         ↪ inks.SearchOrDefault(sequence[i - 1], sequence[i]),
80         ↪ _null))) { }
81     return indexed;
82 }

```

./Sequences/SequencesOptions.cs

```

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

```

```

37 {
38     if (_equalityComparer.Equals(SequenceMarkerLink,
39     ↪ links.Constants.Null))
40     {
41         SequenceMarkerLink = links.CreatePoint();
42     }
43     else
44     {
45         if (!links.Exists(SequenceMarkerLink))
46         {
47             var link = links.CreatePoint();
48             if (!_equalityComparer.Equals(link,
49             ↪ SequenceMarkerLink))
50             {
51                 throw new
52                 ↪ InvalidOperationException("Cannot
53                 ↪ recreate sequence marker link.");
54             }
55         }
56     }
57     if (MarkedSequenceMatcher == null)
58     {
59         MarkedSequenceMatcher = new
60         ↪ MarkedSequenceCreteriaMatcher<TLink>(links,
61         ↪ SequenceMarkerLink);
62     }
63     var balancedVariantConverter = new
64     ↪ BalancedVariantConverter<TLink>(links);
65     if (UseCompression)
66     {
67         if (LinksToSequenceConverter == null)
68         {
69             ICounter<TLink, TLink>
70             ↪ totalSequenceSymbolFrequencyCounter;
71             if (UseSequenceMarker)
72             {
73                 totalSequenceSymbolFrequencyCounter = new
74                 ↪ TotalMarkedSequenceSymbolFrequencyCounter<
75                 ↪ TLink>(links,
76                 ↪ MarkedSequenceMatcher);
77             }
78             else
79             {
80                 totalSequenceSymbolFrequencyCounter = new
81                 ↪ TotalSequenceSymbolFrequencyCounter<TLink>(
82                 ↪ links);
83             }
84             var doubletFrequenciesCache = new
85             ↪ LinkFrequenciesCache<TLink>(links,
86             ↪ totalSequenceSymbolFrequencyCounter);
87             var compressingConverter = new
88             ↪ CompressingConverter<TLink>(links,
89             ↪ balancedVariantConverter,
90             ↪ doubletFrequenciesCache);
91             LinksToSequenceConverter = compressingConverter;
92         }
93     }
94 }

```

```

77     else
78     {
79         if (LinksToSequenceConverter == null)
80         {
81             LinksToSequenceConverter =
82                 ↪ balancedVariantConverter;
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
95                 ↪ collection UseSequenceMarker option must be on.");
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 +
14             ↪ char.MaxValue;
15         public static readonly ulong MapSize = 1 + char.MaxValue;
16
17         private readonly ILinks<ulong> _links;
18         private bool _initialized;
19
20         public UnicodeMap(ILinks<ulong> links) => _links = links;
21
22         public static UnicodeMap InitNew(ILinks<ulong> links)
23         {
24             var map = new UnicodeMap(links);
25             map.Init();
26             return map;
27         }
28
29         public void Init()
30         {
31             if (_initialized)
32             {
33                 return;
34             }
35             _initialized = true;

```

```

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)
45             ↪ transformation meaning, (or infinite amount of
46             ↪ NIL characters before actual Character)
47             var createdLink = _links.CreatePoint();
48             _links.Update(createdLink, firstLink, createdLink);
49             if (createdLink != i)
50             {
51                 throw new InvalidOperationException("Unable to
52                     ↪ initialize UTF 16 table.");
53             }
54         }
55     }
56
57     // 0 - null link
58     // 1 - nil character (0 character)
59     // ...
60     // 65536 (0(1) + 65535 = 65536 possible values)
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     public static ulong FromCharToLink(char character) =>
64         ↪ (ulong)character + 1;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     public static char FromLinkToChar(ulong link) => (char)(link -
68         ↪ 1);
69
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     public static bool IsCharLink(ulong link) => link <= MapSize;
72
73     public static string FromLinksToString(IList<ulong> linksList)
74     {
75         var sb = new StringBuilder();
76         for (int i = 0; i < linksList.Count; i++)
77         {
78             sb.Append(FromLinkToChar(linksList[i]));
79         }
80         return sb.ToString();
81     }
82
83     public static string FromSequenceLinkToString(ulong link,
84         ↪ ILinks<ulong> links)
85     {
86         var sb = new StringBuilder();
87         if (links.Exists(link))
88         {
89             StopableSequenceWalker.WalkRight(link,
90                 ↪ links.GetSource, links.GetTarget,
91                 x => x <= MapSize || links.GetSource(x) == x ||
92                 ↪ links.GetTarget(x) == x, element =>

```



```

86         {
87             sb.Append(FromLinkToChar(element));
88             return true;
89         });
90     }
91     return sb.ToString();
92 }
93
94 public static ulong[] FromCharsToLinkArray(char[] chars) =>
95     ↪ FromCharsToLinkArray(chars, chars.Length);
96
97 public static ulong[] FromCharsToLinkArray(char[] chars, int
98     ↪ 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
121     ↪ sequence)
122 {
123     var result = new List<ulong[]>();
124     var offset = 0;
125     while (offset < sequence.Length)
126     {
127         var currentCategory = CharUnicodeInfo.GetUnicodeCatego
128             ↪ ry(sequence[offset]);
129         var relativeLength = 1;
130         var absoluteLength = offset + relativeLength;
131         while (absoluteLength < sequence.Length &&
132             currentCategory == CharUnicodeInfo.GetUnicodeCa
133                 ↪ tegory(sequence[absoluteLength]))
134         {
135             relativeLength++;
136             absoluteLength++;
137         }
138         // char array to ulong array
139         var innerSequence = new ulong[relativeLength];
140         var maxLength = offset + relativeLength;
141         for (var i = offset; i < maxLength; i++)
142         {
143             innerSequence[i - offset] =
144                 ↪ FromCharToLink(sequence[i]);
145         }
146     }

```

```

140         result.Add(innerSequence);
141         offset += relativeLength;
142     }
143     return result;
144 }
145
146 public static List<ulong[]>
147     ↪ 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 = CharUnicodeInfo.GetUnicodeCa
157                 ↪ tegory(FromLinkToChar(array[offset]));
158             var absoluteLength = offset + relativeLength;
159             while (absoluteLength < array.Length &&
160                 array[absoluteLength] <= LastCharLink &&
161                 currentCategory ==
162                     ↪ CharUnicodeInfo.GetUnicodeCategory(From
163                         ↪ LinkToChar(array[absoluteLength])))
164             {
165                 relativeLength++;
166                 absoluteLength++;
167             }
168         }
169         else
170         {
171             var absoluteLength = offset + relativeLength;
172             while (absoluteLength < array.Length &&
173                 ↪ array[absoluteLength] > LastCharLink)
174             {
175                 relativeLength++;
176                 absoluteLength++;
177             }
178         }
179         // copy array
180         var innerSequence = new ulong[relativeLength];
181         var maxLength = offset + relativeLength;
182         for (var i = offset; i < maxLength; i++)
183         {
184             innerSequence[i - offset] = array[i];
185         }
186         result.Add(innerSequence);
187         offset += relativeLength;
188     }
189     return result;
190 }
191
192 }
193
194 }

```

./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
11     [MethodImpl(MethodImplOptions.AggressiveInlining)]
12     protected override IList<TLink>
13     ↪ GetNextElementAfterPop(IList<TLink> element) =>
14     ↪ Links.GetLink(Links.GetSource(element));
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     protected override IList<TLink>
18     ↪ GetNextElementAfterPush(IList<TLink> element) =>
19     ↪ Links.GetLink(Links.GetTarget(element));
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     protected override IEnumerable<IList<TLink>>
23     ↪ WalkContents(IList<TLink> element)
24     {
25         var start = Links.Constants.IndexPart + 1;
26         for (var i = element.Count - 1; i >= start; i--)
27         {
28             var partLink = Links.GetLink(element[i]);
29             if (IsElement(partLink))
30             {
31                 yield return partLink;
32             }
33         }
34     }
35 }

```

./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
11     [MethodImpl(MethodImplOptions.AggressiveInlining)]
12     protected override IList<TLink>
13     ↪ GetNextElementAfterPop(IList<TLink> element) =>
14     ↪ Links.GetLink(Links.GetTarget(element));
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     protected override IList<TLink>
18     ↪ GetNextElementAfterPush(IList<TLink> element) =>
19     ↪ Links.GetLink(Links.GetSource(element));
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     protected override IEnumerable<IList<TLink>>
23     ↪ WalkContents(IList<TLink> element)
24     {
25         for (var i = Links.Constants.IndexPart + 1; i <
26             ↪ element.Count; i++)
27         {

```

```

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> :
8     ↪ LinksOperatorBase<TLink>, ISequenceWalker<TLink>
9     {
10         // TODO: Use IStack instead of
11         ↪ System.Collections.Generic.Stack, but IStack should
12         ↪ contain IsEmpty property
13         private readonly Stack<IList<TLink>> _stack;
14
15         protected SequenceWalkerBase(ILinks<TLink> links) :
16         ↪ base(links) => _stack = new Stack<IList<TLink>>();
17
18         public IEnumerable<IList<TLink>> Walk(TLink sequence)
19         {
20             if (_stack.Count > 0)
21             {
22                 _stack.Clear(); // This can be replaced with
23                 ↪ while(!_stack.IsEmpty) _stack.Pop()
24             }
25             var element = Links.GetLink(sequence);
26             if (IsElement(element))
27             {
28                 yield return element;
29             }
30             else
31             {
32                 while (true)
33                 {
34                     if (IsElement(element))
35                     {
36                         if (_stack.Count == 0)
37                         {
38                             break;
39                         }
40                         element = _stack.Pop();
41                         foreach (var output in WalkContents(element))
42                         {
43                             yield return output;
44                         }
45                         element = GetNextElementAfterPop(element);
46                     }
47                     else
48                     {
49                         _stack.Push(element);
50                     }
51                 }
52             }
53         }
54     }
55 }

```

```

45         element = GetNextElementAfterPush(element);
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>
57     ⇨ GetNextElementAfterPop(IList<TLink> element);
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected abstract IList<TLink>
61     ⇨ GetNextElementAfterPush(IList<TLink> element);
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected abstract IEnumerable<IList<TLink>>
65     ⇨ WalkContents(IList<TLink> element);
66 }
67 }

```

./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>
9              ⇨ _equalityComparer = 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  }

```

```

36     }
37
38     public void Push(TLink element) => _links.Update(_stack,
39         ⇨ GetStackMarker(), _links.GetOrCreate(GetTop(), element));
40 }

```

./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>
6              ⇨ links, TLink stackMarker)
7          {
8              var stackPoint = links.CreatePoint();
9              var stack = links.Update(stackPoint, stackMarker,
10                  ⇨ stackPoint);
11              return stack;
12          }
13
14          public static void DeleteStack<TLink>(this ILinks<TLink>
15              ⇨ links, TLink stack) => links.Delete(stack);
16      }
17  }

```

./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
12      /// ⇨ for performance improvements.
13      /// TODO: Or even to unfold multiple layers of implementations.
14      /// </remarks>
15      public class SynchronizedLinks<T> : ISynchronizedLinks<T>
16      {
17          public LinksCombinedConstants<T, T, int> Constants { get; }
18          public ISynchronization SyncRoot { get; }
19          public ILinks<T> Sync { get; }
20          public ILinks<T> Unsync { get; }
21
22          public SynchronizedLinks(ILinks<T> links) : this(new
23              ⇨ ReaderWriterLockSynchronization(), links) { }
24
25          public SynchronizedLinks(ISynchronization synchronization,
26              ⇨ ILinks<T> links)
27          {
28              SyncRoot = synchronization;
29              Sync = this;
30              Unsync = links;
31              Constants = links.Constants;
32          }
33      }

```

```

30     public T Count(IList<T> restriction) =>
31     ↪ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
32     public T Each(Func<IList<T>, T> handler, IList<T>
33     ↪ restrictions) => SyncRoot.ExecuteReadOperation(handler,
34     ↪ restrictions, (handler1, restrictions1) =>
35     ↪ Unsync.Each(handler1, restrictions1));
36     public T Create() =>
37     ↪ SyncRoot.ExecuteWriteOperation(Unsync.Create);
38     public T Update(IList<T> restrictions) =>
39     ↪ SyncRoot.ExecuteWriteOperation(restrictions,
40     ↪ Unsync.Update);
41     public void Delete(T link) =>
42     ↪ SyncRoot.ExecuteWriteOperation(link, Unsync.Delete);
43
44     //public T Trigger(IList<T> restriction, Func<IList<T>,
45     ↪ IList<T>, T> matchedHandler, IList<T> substitution,
46     ↪ Func<IList<T>, IList<T>, T> substitutedHandler)
47     //{
48     //    if (restriction != null && substitution != null &&
49     ↪ !substitution.EqualTo(restriction))
50     //        return SyncRoot.ExecuteWriteOperation(restriction,
51     ↪ matchedHandler, substitution, substitutedHandler,
52     ↪ Unsync.Trigger);
53
54     //    return SyncRoot.ExecuteReadOperation(restriction,
55     ↪ matchedHandler, substitution, substitutedHandler,
56     ↪ Unsync.Trigger);
57     //}
58 }

```

./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>,
15     ↪ IReadOnlyList<ulong>, IList<ulong>
16     {
17         private static readonly LinksCombinedConstants<bool, ulong,
18         ↪ int> _constants = Default<LinksCombinedConstants<bool,
19         ↪ ulong, int>>.Instance;
20
21         private const int Length = 3;
22
23         public readonly ulong Index;
24         public readonly ulong Source;
25         public readonly ulong Target;
26
27         public static readonly UInt64Link Null = new UInt64Link();
28

```

```

26     public UInt64Link(params ulong[] values)
27     {
28         Index = values.Length > _constants.IndexPart ?
29         ↪ values[_constants.IndexPart] : _constants.Null;
30         Source = values.Length > _constants.SourcePart ?
31         ↪ values[_constants.SourcePart] : _constants.Null;
32         Target = values.Length > _constants.TargetPart ?
33         ↪ values[_constants.TargetPart] : _constants.Null;
34     }
35
36     public UInt64Link(IList<ulong> values)
37     {
38         Index = values.Count > _constants.IndexPart ?
39         ↪ values[_constants.IndexPart] : _constants.Null;
40         Source = values.Count > _constants.SourcePart ?
41         ↪ values[_constants.SourcePart] : _constants.Null;
42         Target = values.Count > _constants.TargetPart ?
43         ↪ values[_constants.TargetPart] : _constants.Null;
44     }
45
46     public UInt64Link(ulong index, ulong source, ulong target)
47     {
48         Index = index;
49         Source = source;
50         Target = target;
51     }
52
53     public UInt64Link(ulong source, ulong target)
54     : this(_constants.Null, source, target)
55     {
56         Source = source;
57         Target = target;
58     }
59
60     public static UInt64Link Create(ulong source, ulong target) =>
61     ↪ new UInt64Link(source, target);
62
63     public override int GetHashCode() => (Index, Source,
64     ↪ Target).GetHashCode();
65
66     public bool IsNull() => Index == _constants.Null
67     ↪ && Source == _constants.Null
68     ↪ && Target == _constants.Null;
69
70     public override bool Equals(object other) => other is
71     ↪ UInt64Link && Equals((UInt64Link)other);
72
73     public bool Equals(UInt64Link other) => Index == other.Index
74     ↪ && Source == other.Source
75     ↪ && Target == other.Target;
76
77     public static string ToString(ulong index, ulong source, ulong
78     ↪ target) => $"({index}: {source}->{target})";
79
80     public static string ToString(ulong source, ulong target) =>
81     ↪ $"({source}->{target})";
82
83     public static implicit operator ulong[] (UInt64Link link) =>
84     ↪ link.ToArray();
85

```

```

74     public static implicit operator UInt64Link(ulong[] linkArray)
75     ↪ => new UInt64Link(linkArray);
76
77     public ulong[] ToArray()
78     {
79         var array = new ulong[Length];
80         CopyTo(array, 0);
81         return array;
82     }
83
84     public override string ToString() => Index == _constants.Null
85     ↪ ? ToString(Source, Target) : ToString(Index, Source,
86     ↪ Target);
87
88     #region IList
89
90     public ulong this[int index]
91     {
92         get
93         {
94             Ensure.Always.ArgumentInRange(index, new Range<int>(0,
95             ↪ Length - 1), nameof(index));
96             if (index == _constants.IndexPart)
97             {
98                 return Index;
99             }
100             if (index == _constants.SourcePart)
101             {
102                 return Source;
103             }
104             if (index == _constants.TargetPart)
105             {
106                 return Target;
107             }
108             throw new NotSupportedException(); // Impossible path
109             ↪ due to Ensure.ArgumentInRange
110         }
111         set => throw new NotSupportedException();
112     }
113
114     public int Count => Length;
115
116     public bool IsReadOnly => true;
117
118     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
119
120     public IEnumerator<ulong> GetEnumerator()
121     {
122         yield return Index;
123         yield return Source;
124         yield return Target;
125     }
126
127     public void Add(ulong item) => throw new
128     ↪ NotSupportedException();
129
130     public void Clear() => throw new NotSupportedException();
131
132     public bool Contains(ulong item) => IndexOf(item) >= 0;
133
134     public void CopyTo(ulong[] array, int arrayIndex)

```

```

129     {
130         Ensure.Always.ArgumentNotNull(array, nameof(array));
131         Ensure.Always.ArgumentInRange(arrayIndex, new
132         ↪ Range<int>(0, array.Length - 1), nameof(arrayIndex));
133         if (arrayIndex + Length > array.Length)
134         {
135             throw new ArgumentException();
136         }
137         array[arrayIndex++] = Index;
138         array[arrayIndex++] = Source;
139         array[arrayIndex] = Target;
140     }
141
142     public bool Remove(ulong item) =>
143     ↪ Throw.A.NotSupportedExceptionAndReturn<bool>();
144
145     public int IndexOf(ulong item)
146     {
147         if (Index == item)
148         {
149             return _constants.IndexPart;
150         }
151         if (Source == item)
152         {
153             return _constants.SourcePart;
154         }
155         if (Target == item)
156         {
157             return _constants.TargetPart;
158         }
159         return -1;
160     }
161
162     public void Insert(int index, ulong item) => throw new
163     ↪ NotSupportedException();
164
165     public void RemoveAt(int index) => throw new
166     ↪ NotSupportedException();
167
168     #endregion
169 }

```

./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>
14             ↪ Constants = Default<LinksCombinedConstants<bool,
15             ↪ ulong, int>>.Instance;
16
17         public static void UseUnicode(this ILinks<ulong> links) =>
18             ↪ UnicodeMap.InitNew(links);
19
20         public static void EnsureEachLinkExists(this ILinks<ulong>
21             ↪ links, IList<ulong> sequence)
22         {
23             if (sequence == null)
24             {
25                 return;
26             }
27             for (var i = 0; i < sequence.Count; i++)
28             {
29                 if (!links.Exists(sequence[i]))
30                 {
31                     throw new ArgumentLinkDoesNotExistsException<ulong>
32                     ↪ >(sequence[i],
33                     ↪ $"sequence[{i}]");
34                 }
35             }
36
37         public static void EnsureEachLinkIsAnyOrExists(this
38             ↪ ILinks<ulong> links, IList<ulong> sequence)
39         {
40             if (sequence == null)
41             {
42                 return;
43             }
44             for (var i = 0; i < sequence.Count; i++)
45             {
46                 if (sequence[i] != Constants.Any &&
47                     ↪ !links.Exists(sequence[i]))
48                 {
49                     throw new ArgumentLinkDoesNotExistsException<ulong>
50                     ↪ >(sequence[i],
51                     ↪ $"sequence[{i}]");
52                 }
53             }
54
55         public static bool AnyLinkIsAny(this ILinks<ulong> links,
56             ↪ params ulong[] sequence)
57         {
58             if (sequence == null)
59             {
60                 return false;
61             }
62         }
63     }
64 }

```

```

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

```

```

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
108                 ↳ UInt64Link(links.GetLink(link.Source));
109             if (isElement(source))
110             {
111                 appendElement(sb, source);
112             }
113             else
114             {
115                 links.AppendStructure(sb, visited,
116                     ↳ source.Index, isElement,
117                     ↳ appendElement, renderIndex);
118             }
119         }
120         sb.Append(' ');
121         if (link.Target == link.Index)
122         {
123             sb.Append(link.Index);
124         }
125         else
126         {
127             var target = new
128                 ↳ UInt64Link(links.GetLink(link.Target));
129             if (isElement(target))
130             {
131                 appendElement(sb, target);
132             }
133             else
134             {
135                 links.AppendStructure(sb, visited,
136                     ↳ target.Index, isElement,
137                     ↳ appendElement, renderIndex);
138             }
139         }
140         sb.Append(')');
141     }
142     else
143     {
144         if (renderDebug)
145         {
146             sb.Append('*');
147         }
148         sb.Append(linkIndex);
149     }
150 }
151
152 if (renderIndex)
153 {
154     sb.Append(link.Index);
155     sb.Append(':');
156 }
157 if (link.Source == link.Index)
158 {
159     sb.Append(link.Index);
160 }
161 else
162 {
163     var source = new
164         ↳ UInt64Link(links.GetLink(link.Source));
165     if (isElement(source))
166     {
167         appendElement(sb, source);
168     }
169     else
170     {
171         links.AppendStructure(sb, visited,
172             ↳ source.Index, isElement,
173             ↳ appendElement, renderIndex);
174     }
175 }
176 sb.Append(' ');
177 if (link.Target == link.Index)
178 {
179     sb.Append(link.Index);
180 }
181 else
182 {
183     var target = new
184         ↳ UInt64Link(links.GetLink(link.Target));
185     if (isElement(target))
186     {
187         appendElement(sb, target);
188     }
189     else
190     {
191         links.AppendStructure(sb, visited,
192             ↳ target.Index, isElement,
193             ↳ appendElement, renderIndex);
194     }
195 }
196 sb.Append(')');
197 }
198 else
199 {
200     if (renderDebug)
201     {
202         sb.Append('*');
203     }
204     sb.Append(linkIndex);
205 }
206 }
207
208 if (renderIndex)
209 {
210     sb.Append(link.Index);
211     sb.Append(':');
212 }
213 if (link.Source == link.Index)
214 {
215     sb.Append(link.Index);
216 }
217 else
218 {
219     var source = new
220         ↳ UInt64Link(links.GetLink(link.Source));
221     if (isElement(source))
222     {
223         appendElement(sb, source);
224     }
225     else
226     {
227         links.AppendStructure(sb, visited,
228             ↳ source.Index, isElement,
229             ↳ appendElement, renderIndex);
230     }
231 }
232 sb.Append(' ');
233 if (link.Target == link.Index)
234 {
235     sb.Append(link.Index);
236 }
237 else
238 {
239     var target = new
240         ↳ UInt64Link(links.GetLink(link.Target));
241     if (isElement(target))
242     {
243         appendElement(sb, target);
244     }
245     else
246     {
247         links.AppendStructure(sb, visited,
248             ↳ target.Index, isElement,
249             ↳ appendElement, renderIndex);
250     }
251 }
252 sb.Append(')');
253 }
254 else
255 {
256     if (renderDebug)
257     {
258         sb.Append('*');
259     }
260     sb.Append(linkIndex);
261 }
262 }

```

```

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 :
17         ↳ LinksDisposableDecoratorBase<ulong> //-V3073
18     {
19         /// <remarks>
20         /// Альтернативные варианты хранения трансформации (элемента
21         /// транзакции):
22         /// private enum TransitionType
23         /// {
24         ///     Creation,
25         ///     UpdateOf,
26         ///     UpdateTo,
27         ///     Deletion
28         /// }
29         /// private struct Transition
30         /// {
31         ///     public ulong TransactionId;
32         ///     public UniqueTimestamp Timestamp;
33         ///     public TransactionItemType Type;
34         ///     public Link Source;
35         ///     public Link Linker;
36         ///     public Link Target;
37         /// }
38         /// Или
39         /// public struct TransitionHeader
40         /// {
41         ///     public ulong TransactionIdCombined;
42         ///     public ulong TimestampCombined;
43         /// }
44         /// public ulong TransactionId
45         /// {
46         ///     get
47         ///     {
48         ///
49         ///

```

```

50     ///         return (ulong) mask & TransactionIdCombined;
51     ///     }
52     /// }
53     ///
54     /// public UniqueTimestamp Timestamp
55     /// {
56     ///     get
57     ///     {
58     ///         return (UniqueTimestamp)mask &
59     ///         TransactionIdCombined;
60     ///     }
61     /// }
62     /// public TransactionItemType Type
63     /// {
64     ///     get
65     ///     {
66     ///         // Использовать по одному биту из
67     ///         TransactionId и Timestamp,
68     ///         // для значения в 2 бита, которое представляет
69     ///         // тип операции
70     ///         throw new NotImplementedException();
71     ///     }
72     /// }
73     /// private struct Transition
74     /// {
75     ///     public TransitionHeader Header;
76     ///     public Link Source;
77     ///     public Link Linker;
78     ///     public Link Target;
79     /// }
80     ///
81     /// </remarks>
82     public struct Transition
83     {
84         public static readonly long Size =
85             ↳ StructureHelpers.SizeOf<Transition>();
86
87         public readonly ulong TransactionId;
88         public readonly UInt64Link Before;
89         public readonly UInt64Link After;
90         public readonly Timestamp Timestamp;
91
92         public Transition(UniqueTimestampFactory
93             ↳ uniqueTimestampFactory, ulong transactionId,
94             ↳ UInt64Link before, UInt64Link after)
95         {
96             TransactionId = transactionId;
97             Before = before;
98             After = after;
99             Timestamp = uniqueTimestampFactory.Create();
100         }
101
102         public Transition(UniqueTimestampFactory
103             ↳ uniqueTimestampFactory, ulong transactionId,
104             ↳ UInt64Link before)
105             : this(uniqueTimestampFactory, transactionId, before,
106                 ↳ default)

```

```

101     {
102     }
103
104     public Transition(UniqueTimestampFactory
105         ↳ uniqueTimestampFactory, ulong transactionId)
106         : this(uniqueTimestampFactory, transactionId, default,
107             ↳ default)
108     {
109     }
110
111     public override string ToString() => $"{Timestamp}
112         ↳ {TransactionId}: {Before} => {After}";
113     }
114
115     /// <remarks>
116     /// Другие варианты реализации транзакций (атомарности):
117     /// 1. Разделение хранения значения связи ((Source Target)
118     ///     ↳ или (Source Linker Target)) и индексов.
119     /// 2. Хранение трансформаций/операций в отдельном
120     ///     ↳ хранилище Links, но дополнительно потребуется решить вопрос
121     ///     ↳ со ссылками на внешние идентификаторы, или как-то
122     ///     ↳ иначе решить вопрос с пересечениями идентификаторов.
123     ///
124     /// Где хранить промежуточный список транзакций?
125     ///
126     /// В оперативной памяти:
127     /// Минусы:
128     /// 1. Может усложнить систему, если она будет
129     ///     ↳ функционировать самостоятельно,
130     ///     ↳ так как нужно отдельно выделять память под список
131     ///     ↳ трансформаций.
132     /// 2. Выделенной оперативной памяти может не хватить, в
133     ///     ↳ том случае,
134     ///     ↳ если транзакция использует слишком много трансформаций.
135     ///     ↳ -> Можно использовать жёсткий диск для слишком
136     ///     ↳ длинных транзакций.
137     ///     ↳ -> Максимальный размер списка трансформаций можно
138     ///     ↳ ограничить / задать константой.
139     /// 3. При подтверждении транзакции (Commit) все
140     ///     ↳ трансформации записываются разом создавая задержку.
141     ///
142     /// На жёстком диске:
143     /// Минусы:
144     /// 1. Длительный отклик, на запись каждой трансформации.
145     /// 2. Лог транзакций дополнительно наполняется
146     ///     ↳ отменёнными транзакциями.
147     ///     ↳ -> Это может решаться упаковкой/исключением
148     ///     ↳ дублирующих операций.
149     ///     ↳ -> Также это может решаться тем, что короткие
150     ///     ↳ транзакции вообще
151     ///     ↳ не будут записываться в случае отката.
152     /// 3. Перед тем как выполнять отмену операций транзакции
153     ///     ↳ нужно дождаться пока все операции (трансформации)
154     ///     ↳ будут записаны в лог.
155     ///
156     /// </remarks>
157     public class Transaction : DisposableBase
158     {

```



```

143 private readonly Queue<Transition> _transitions;
144 private readonly UInt64LinksTransactionsLayer _layer;
145 public bool IsCommitted { get; private set; }
146 public bool IsReverted { get; private set; }
147
148 public Transaction(UInt64LinksTransactionsLayer layer)
149 {
150     _layer = layer;
151     if (_layer._currentTransactionId != 0)
152     {
153         throw new NotSupportedException("Nested
154             ↳ transactions not supported.");
155     }
156     IsCommitted = false;
157     IsReverted = false;
158     _transitions = new Queue<Transition>();
159     SetCurrentTransaction(layer, this);
160 }
161
162 public void Commit()
163 {
164     EnsureTransactionAllowsWriteOperations(this);
165     while (_transitions.Count > 0)
166     {
167         var transition = _transitions.Dequeue();
168         _layer._transitions.Enqueue(transition);
169     }
170     _layer._lastCommittedTransactionId =
171         ↳ _layer._currentTransactionId;
172     IsCommitted = true;
173 }
174
175 private void Revert()
176 {
177     EnsureTransactionAllowsWriteOperations(this);
178     var transitionsToRevert = new
179         ↳ Transition[_transitions.Count];
180     _transitions.CopyTo(transitionsToRevert, 0);
181     for (var i = transitionsToRevert.Length - 1; i >= 0;
182         ↳ i--)
183     {
184         _layer.RevertTransition(transitionsToRevert[i]);
185     }
186     IsReverted = true;
187 }
188
189 public static void
190     ↳ SetCurrentTransaction(UInt64LinksTransactionsLayer
191     ↳ layer, Transaction transaction)
192 {
193     layer._currentTransactionId =
194         ↳ layer._lastCommittedTransactionId + 1;
195     layer._currentTransactionTransitions =
196         ↳ transaction._transitions;
197     layer._currentTransaction = transaction;
198 }
199
200 public static void
201     ↳ EnsureTransactionAllowsWriteOperations(Transaction
202     ↳ transaction)

```

```

193 {
194     if (transaction.IsReverted)
195     {
196         throw new InvalidOperationException("Transation is
197             ↳ reverted.");
198     }
199     if (transaction.IsCommitted)
200     {
201         throw new InvalidOperationException("Transation is
202             ↳ committed.");
203     }
204 }
205
206 protected override void DisposeCore(bool manual, bool
207     ↳ wasDisposed)
208 {
209     if (!wasDisposed && _layer != null &&
210         ↳ !_layer.IsDisposed)
211     {
212         if (!IsCommitted && !IsReverted)
213         {
214             Revert();
215         }
216         _layer.ResetCurrentTransation();
217     }
218 }
219
220 // TODO: THIS IS EXCEPTION WORKAROUND, REMOVE IT THEN
221 ↳ https://github.com/linksplatform/Disposables/issues/13
222 ↳ FIXED
223 protected override bool AllowMultipleDisposeCalls => true;
224 }
225
226 public static readonly TimeSpan DefaultPushDelay =
227     ↳ TimeSpan.FromSeconds(0.1);
228
229 private readonly string _logAddress;
230 private readonly FileStream _log;
231 private readonly Queue<Transition> _transitions;
232 private readonly UniqueTimestampFactory
233     ↳ _uniqueTimestampFactory;
234 private Task _transitionsPusher;
235 private Transition _lastCommittedTransition;
236 private ulong _currentTransactionId;
237 private Queue<Transition> _currentTransactionTransitions;
238 private Transaction _currentTransaction;
239 private ulong _lastCommittedTransactionId;
240
241 public UInt64LinksTransactionsLayer(ILinks<ulong> links,
242     ↳ string logAddress)
243     : base(links)
244 {
245     if (string.IsNullOrEmpty(logAddress))
246     {
247         throw new ArgumentNullException(nameof(logAddress));
248     }
249     // В первой строке файла хранится последняя закоммиченную
250     ↳ транзакцию.
251     // При запуске это используется для проверки удачного
252     ↳ закрытия файла лога.

```

```

242 // In the first line of the file the last committed
    ↳ transaction is stored.
243 // On startup, this is used to check that the log file is
    ↳ successfully closed.
244 var lastCommittedTransition =
    ↳ FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
245 var lastWrittenTransition =
    ↳ FileHelpers.ReadLastOrDefault<Transition>(logAddress);
246 if (!lastCommittedTransition.Equals(lastWrittenTransition))
247 {
248     Dispose();
249     throw new NotSupportedException("Database is damaged,
        ↳ autorecovery is not supported yet.");
250 }
251 if (lastCommittedTransition.Equals(default(Transition)))
252 {
253     FileHelpers.WriteFirst(logAddress,
        ↳ lastCommittedTransition);
254 }
255 _lastCommittedTransition = lastCommittedTransition;
256 // TODO: Think about a better way to calculate or store
    ↳ this value
257 var allTransitions =
    ↳ FileHelpers.ReadAll<Transition>(logAddress);
258 _lastCommittedTransactionId = allTransitions.Max(x =>
    ↳ x.TransactionId);
259 _uniqueTimestampFactory = new UniqueTimestampFactory();
260 _logAddress = logAddress;
261 _log = FileHelpers.Append(logAddress);
262 _transitions = new Queue<Transition>();
263 _transitionsPusher = new Task(TransitionsPusher);
264 _transitionsPusher.Start();
265 }
266
267 public IList<ulong> GetLinkValue(ulong link) =>
    ↳ Links.GetLink(link);
268
269 public override ulong Create()
270 {
271     var createdLinkIndex = Links.Create();
272     var createdLink = new
        ↳ UInt64Link(Links.GetLink(createdLinkIndex));
273     CommitTransition(new Transition(_uniqueTimestampFactory,
        ↳ _currentTransactionId, default, createdLink));
274     return createdLinkIndex;
275 }
276
277 public override ulong Update(IList<ulong> parts)
278 {
279     var beforeLink = new
        ↳ UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
280     parts[Constants.IndexPart] = Links.Update(parts);
281     var afterLink = new
        ↳ UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
282     CommitTransition(new Transition(_uniqueTimestampFactory,
        ↳ _currentTransactionId, 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(_cu_
            ↳ rrentTransaction);
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         _log.Write(transition);
339         _lastCommittedTransition = transition;
340     }
341 }
342
343 private void TransitionsPusher()
344 {
345     while (!IsDisposed && _transitionsPusher != null)
346     {
347         Thread.Sleep(DefaultPushDelay);
348         PushTransitions();
349     }
350 }
351
352 public Transaction BeginTransaction() => new Transaction(this);
353
354 private void DisposeTransitions()
355 {
356     try
357     {
358         var pusher = _transitionsPusher;
359         if (pusher != null)
360         {
361             _transitionsPusher = null;
362             pusher.Wait();
363         }
364         if (_transitions != null)
365

```

```

366     {
367         PushTransitions();
368     }
369     Disposable.TryDispose(_log);
370     FileHelpers.WriteFirst(_logAddress,
371         ↪ _lastCommittedTransition);
372 }
373 catch
374 {
375 }
376
377 #region DisposalBase
378
379 protected override void DisposeCore(bool manual, bool
380     ↪ wasDisposed)
381 {
382     if (!wasDisposed)
383     {
384         DisposeTransitions();
385     }
386     base.DisposeCore(manual, wasDisposed);
387 }
388
389 #endregion
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, 4
./Decorators/LinksDependenciesValidator.cs, 4
./Decorators/LinksDisposableDecoratorBase.cs, 4
./Decorators/LinksInnerReferenceValidator.cs, 5
./Decorators/LinksNonExistentReferencesCreator.cs, 5
./Decorators/LinksNullToSelfReferenceResolver.cs, 5
./Decorators/LinksSelfReferenceResolver.cs, 6
./Decorators/LinksUniquenessResolver.cs, 6
./Decorators/LinksUniquenessValidator.cs, 6
./Decorators/NonNullContentsLinkDeletionResolver.cs, 7
./Decorators/UInt64Links.cs, 7
./Decorators/UniLinks.cs, 8
./Doublet.cs, 12
./DoubletComparer.cs, 11
./Hybrid.cs, 12
./ILinks.cs, 13
./ILinksExtensions.cs, 13
./ISynchronizedLinks.cs, 21
./Incrementers/FrequencyIncrementer.cs, 20
./Incrementers/LinkFrequencyIncrementer.cs, 20
./Incrementers/UnaryNumberIncrementer.cs, 21
./Link.cs, 21
./LinkExtensions.cs, 23
./LinksOperatorBase.cs, 23
./PropertyOperators/DefaultLinkPropertyOperator.cs, 23
./PropertyOperators/FrequencyPropertyOperator.cs, 24
./ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs, 31
./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs, 32
./ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 24
./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs, 42
./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs, 42
./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 37
./Sequences/Converters/BalancedVariantConverter.cs, 47
./Sequences/Converters/CompressingConverter.cs, 47
./Sequences/Converters/LinksListToSequenceConverterBase.cs, 50
./Sequences/Converters/OptimalVariantConverter.cs, 50
./Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 51
./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs, 51
./Sequences/CreteriaMatchers/MarkedSequenceCreteriaMatcher.cs, 52
./Sequences/DefaultSequenceAppender.cs, 52
./Sequences/DuplicateSegmentsCounter.cs, 52
./Sequences/DuplicateSegmentsProvider.cs, 53
./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs, 54
./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToltsFrequencyNumberConverter.cs, 54
./Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 55
./Sequences/Frequencies/Cache/LinkFrequency.cs, 56
./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 56
./Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 56
./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 57
./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 57
./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 57
./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 57
./Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 58
./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 59
./Sequences/HeightProviders/ISequenceHeightProvider.cs, 59
./Sequences/Sequences.Experiments.ReadSequence.cs, 85
./Sequences/Sequences.Experiments.cs, 66
./Sequences/Sequences.cs, 59
./Sequences/SequencesExtensions.cs, 86
./Sequences/SequencesIndexer.cs, 86
./Sequences/SequencesOptions.cs, 87
./Sequences/UnicodeMap.cs, 88
./Sequences/Walkers/LeftSequenceWalker.cs, 89
./Sequences/Walkers/RightSequenceWalker.cs, 90
./Sequences/Walkers/SequenceWalkerBase.cs, 90
./Stacks/Stack.cs, 91
./Stacks/StackExtensions.cs, 91
./SynchronizedLinks.cs, 91
./UInt64Link.cs, 92
./UInt64LinkExtensions.cs, 93
./UInt64LinksExtensions.cs, 93
./UInt64LinksTransactionsLayer.cs, 95
./obj/Debug/netstandard2.0/Platform.Data.Doublets.AssemblyInfo.cs, 23