

LinksPlatform's Platform.Data.Doublets Class Library

./Converters/AddressToUnaryNumberConverter.cs

```

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

```

./Converters/LinkToItsFrequencyNumberConveter.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Converters
6  {
7      public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
8          ⇨ IConverter<Doublet<TLink>, TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ⇨ EqualityComparer<TLink>.Default;
12
13         private readonly ISpecificPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
14         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
15
16         public LinkToItsFrequencyNumberConveter(
17             ILinks<TLink> links,
18             ISpecificPropertyOperator<TLink, TLink> frequencyPropertyOperator,
19             IConverter<TLink> unaryNumberToAddressConverter)
20             : base(links)
21         {
22             _frequencyPropertyOperator = frequencyPropertyOperator;
23             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
24         }
25
26         public TLink Convert(Doublet<TLink> doublet)
27         {
28             var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
29             if (_equalityComparer.Equals(link, Links.Constants.Null))
30             {
31                 throw new ArgumentException($"Link with {doublet.Source} source and
32                     ⇨ {doublet.Target} target not found.", nameof(doublet));
33             }
34         }
35     }
36 }

```

```

31         var frequency = _frequencyPropertyOperator.Get(link);
32         if (_equalityComparer.Equals(frequency, default))
33         {
34             return default;
35         }
36         var frequencyNumber = Links.GetSource(frequency);
37         var number = _unaryNumberToAddressConverter.Convert(frequencyNumber);
38         return number;
39     }
40 }
41 }

```

./Converters/PowerOf2ToUnaryNumberConverter.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Converters
6  {
7      public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
8          ⇨ IConverter<int, TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ⇨ EqualityComparer<TLink>.Default;
12
13         private readonly TLink[] _unaryNumberPowersOf2;
14
15         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
16         {
17             _unaryNumberPowersOf2 = new TLink[64];
18             _unaryNumberPowersOf2[0] = one;
19         }
20
21         public TLink Convert(int power)
22         {
23             if (power < 0 || power >= _unaryNumberPowersOf2.Length)
24             {
25                 throw new ArgumentOutOfRangeException(nameof(power));
26             }
27             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
28             {
29                 return _unaryNumberPowersOf2[power];
30             }
31             var previousPowerOf2 = Convert(power - 1);
32             var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
33             _unaryNumberPowersOf2[power] = powerOf2;
34             return powerOf2;
35         }
36     }
37 }

```

./Converters/UnaryNumberToAddressAddOperationConverter.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4
5  namespace Platform.Data.Doublets.Converters
6  {
7      public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
8          ⇨ IConverter<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ⇨ EqualityComparer<TLink>.Default;
12
13         private Dictionary<TLink, TLink> _unaryToUInt64;
14         private readonly TLink _unaryOne;
15
16         public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
17             : base(links)
18         {
19             _unaryOne = unaryOne;
20             InitUnaryToUInt64();
21         }
22
23         private void InitUnaryToUInt64()
24         {
25             _unaryToUInt64 = new Dictionary<TLink, TLink>
26             {
27                 { _unaryOne, Integer<TLink>.One }
28             };
29         }
30     }
31 }

```

```

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 public TLink Convert(TLink unaryNumber)
36 {
37     if (_equalityComparer.Equals(unaryNumber, default))
38     {
39         return default;
40     }
41     if (_equalityComparer.Equals(unaryNumber, _unaryOne))
42     {
43         return Integer<TLink>.One;
44     }
45     var source = Links.GetSource(unaryNumber);
46     var target = Links.GetTarget(unaryNumber);
47     if (_equalityComparer.Equals(source, target))
48     {
49         return _unaryToUInt64[unaryNumber];
50     }
51     else
52     {
53         var result = _unaryToUInt64[source];
54         TLink lastValue;
55         while (!_unaryToUInt64.TryGetValue(target, out lastValue))
56         {
57             source = Links.GetSource(target);
58             result = ArithmeticHelpers.Add(result, _unaryToUInt64[source]);
59             target = Links.GetTarget(target);
60         }
61         result = ArithmeticHelpers.Add(result, lastValue);
62         return result;
63     }
64 }
65 }
66 }

```

./Converters/UnaryNumberToAddressOrOperationConverter.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Reflection;
4  using Platform.Numbers;
5
6  namespace Platform.Data.Doublets.Converters
7  {
8      public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
            ↪ IConverter<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
11
12         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
13
14         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
            ↪ TLink> powerOf2ToUnaryNumberConverter)
            : base(links)
15         {
16             _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
17             for (int i = 0; i < CachedTypeInfo<TLink>.BitsLength; i++)
18             {
19                 _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
20             }
21         }
22
23         public TLink Convert(TLink sourceNumber)
24         {
25             var source = sourceNumber;
26             var target = Links.Constants.Null;
27             while (!_equalityComparer.Equals(source, Links.Constants.Null))
28             {
29                 if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
30                 {
31                     source = Links.Constants.Null;
32                 }
33             }

```

```

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

```

./Decorators/LinksCascadeDependenciesResolver.cs

```

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

```

./Decorators/LinksCascadeUniquenessAndDependenciesResolver.cs

```

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

```

```

18     var references = ArrayPool.Allocate<TLink>((long)(referencesAsSourceCount +
19         ↪ referencesAsTargetCount));
20     var referencesFiller = new ArrayFiller<TLink, TLink>(references, Constants.Continue);
21     Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any,
22         ↪ oldLinkAddress, Constants.Any);
23     Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, Constants.Any,
24         ↪ oldLinkAddress);
25     for (ulong i = 0; i < referencesAsSourceCount; i++)
26     {
27         var reference = references[i];
28         if (!_equalityComparer.Equals(reference, oldLinkAddress))
29         {
30             Links.Update(reference, newLinkAddress, Links.GetTarget(reference));
31         }
32     }
33     for (var i = (long)referencesAsSourceCount; i < references.Length; i++)
34     {
35         var reference = references[i];
36         if (!_equalityComparer.Equals(reference, oldLinkAddress))
37         {
38             Links.Update(reference, Links.GetSource(reference), newLinkAddress);
39         }
40     }
41     ArrayPool.Free(references);
42     return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);

```

./Decorators/LinksDecoratorBase.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Data.Constants;
4
5  namespace Platform.Data.Doublets.Decorators
6  {
7      public abstract class LinksDecoratorBase<T> : ILinks<T>
8      {
9          public LinksCombinedConstants<T, T, int> Constants { get; }
10
11          public readonly ILinks<T> Links;
12
13          protected LinksDecoratorBase(ILinks<T> links)
14          {
15              Links = links;
16              Constants = links.Constants;
17          }
18
19          public virtual T Count(IList<T> restriction) => Links.Count(restriction);
20
21          public virtual T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
22              ↪ Links.Each(handler, restrictions);
23
24          public virtual T Create() => Links.Create();
25
26          public virtual T Update(IList<T> restrictions) => Links.Update(restrictions);
27
28          public virtual void Delete(T link) => Links.Delete(link);
29      }

```

./Decorators/LinksDependenciesValidator.cs

```

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

```

```

18         base.Delete(link);
19     }
20 }
21 }

```

./Decorators/LinksDisposableDecoratorBase.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Disposables;
4  using Platform.Data.Constants;
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public abstract class LinksDisposableDecoratorBase<T> : DisposableBase, ILinks<T>
9      {
10         public LinksCombinedConstants<T, T, int> Constants { get; }
11
12         public readonly ILinks<T> Links;
13
14         protected LinksDisposableDecoratorBase(ILinks<T> links)
15         {
16             Links = links;
17             Constants = links.Constants;
18         }
19
20         public virtual T Count(IList<T> restriction) => Links.Count(restriction);
21
22         public virtual T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
23             ↪ Links.Each(handler, restrictions);
24
25         public virtual T Create() => Links.Create();
26
27         public virtual T Update(IList<T> restrictions) => Links.Update(restrictions);
28
29         public virtual void Delete(T link) => Links.Delete(link);
30
31         protected override bool AllowMultipleDisposeCalls => true;
32
33         protected override void DisposeCore(bool manual, bool wasDisposed) =>
34             ↪ Disposable.TryDispose(Links);
35     }
36 }

```

./Decorators/LinksInnerReferenceValidator.cs

```

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

```

```

36     }
37 }

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

```

```

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

```

```

./Decorators/LinksSelfReferenceResolver.cs
1  using System;
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets.Decorators
5  {
6      public class LinksSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
7      {
8          private static readonly EqualityComparer<TLink> _equalityComparer =
9              ↪ EqualityComparer<TLink>.Default;
10
11         public LinksSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
12
13         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
14         {
15             if (!_equalityComparer.Equals(Constants.Any, Constants.Itself)
16                 && ((restrictions.Count > Constants.IndexPart) &&
17                     ↪ _equalityComparer.Equals(restrictions[Constants.IndexPart], Constants.Itself))
18                 || ((restrictions.Count > Constants.SourcePart) &&
19                     ↪ _equalityComparer.Equals(restrictions[Constants.SourcePart], Constants.Itself))

```

```

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

```

./Decorators/LinksUniquenessResolver.cs

```

1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Decorators
4 {
5     public class LinksUniquenessResolver<TLink> : LinksDecoratorBase<TLink>
6     {
7         private static readonly EqualityComparer<TLink> _equalityComparer =
8         ↪ EqualityComparer<TLink>.Default;
9
10        public LinksUniquenessResolver(ICollection<TLink> links) : base(links) { }
11
12        public override TLink Update(ICollection<TLink> restrictions)
13        {
14            var newLinkAddress = Links.SearchOrDefault(restrictions[Constants.SourcePart],
15            ↪ restrictions[Constants.TargetPart]);
16            if (_equalityComparer.Equals(newLinkAddress, default))
17            {
18                return base.Update(restrictions);
19            }
20            return ResolveAddressChangeConflict(restrictions[Constants.IndexPart],
21            ↪ newLinkAddress);
22        }
23
24        protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
25        ↪ newLinkAddress)
26        {
27            if (Links.Exists(oldLinkAddress))
28            {
29                Delete(oldLinkAddress);
30            }
31            return newLinkAddress;
32        }
33    }
34 }
35 }

```

./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(ICollection<T> links) : base(links) { }
8
9         public override T Update(ICollection<T> restrictions)
10        {
11            Links.EnsureDoesNotExists(restrictions[Constants.SourcePart],
12            ↪ restrictions[Constants.TargetPart]);
13            return base.Update(restrictions);
14        }
15    }
16 }

```


./Decorators/NonNullContentsLinkDeletionResolver.cs

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

./Decorators/UInt64Links.cs

```
1 using System;
2 using System.Collections.Generic;
3 using Platform.Collections;
4 using Platform.Collections.Arrays;
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     /// <summary>
9     /// Представляет объект для работы с базой данных (файлом) в формате Links (массива
10     /// ↪ взаимосвязей).
11     /// </summary>
12     /// <remarks>
13     /// Возможные оптимизации:
14     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
15     ///     + меньше объём БД
16     ///     - меньше производительность
17     ///     - больше ограничение на количество связей в БД)
18     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
19     ///     + меньше объём БД
20     ///     - больше сложность
21     ///
22     /// AVL - высота дерева может позволить точно рассчитать размер дерева, нет необходимости
23     /// ↪ в SBT.
24     /// AVL дерево можно прошить.
25     ///
26     /// Текущее теоретическое ограничение на размер связей - long.MaxValue
27     /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
28     /// ↪ (битовыми строками) - вариант матрицы (выстраиваемой лениво).
29     ///
30     /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
31     /// ↪ выбрасываться только при #if DEBUG
32     /// </remarks>
33     public class UInt64Links : LinksDisposableDecoratorBase<ulong>
34     {
35         public UInt64Links(ILinks<ulong> links) : base(links) { }
36
37         public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
38         {
39             this.EnsureLinkIsAnyOrExists(restrictions);
40             return Links.Each(handler, restrictions);
41         }
42
43         public override ulong Create() => Links.CreatePoint();
44
45         public override ulong Update(IList<ulong> restrictions)
46         {
47             if (restrictions.IsNullOrEmpty())
48             {
49                 return Constants.Null;
50             }
51             // TODO: Remove usages of these hacks (these should not be backwards compatible)
52             if (restrictions.Count == 2)
53             {
54                 return this.Merge(restrictions[0], restrictions[1]);
55             }
56             if (restrictions.Count == 4)
57             {
58                 return this.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
59                     ↪ restrictions[2], restrictions[3]);
60             }
61             // TODO: Looks like this is a common type of exceptions linked with restrictions
62             ↪ support
63             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] !=
76             ↪ newTarget)
77         {
78             Links.Update(updatedLink, newSource == Constants.Itself ? updatedLink :
79                 ↪ newSource,
80                 newTarget == Constants.Itself ? updatedLink :
81                 ↪ newTarget);
82         }
83         return updatedLink;
84     }
85     else
86     {
87         // Replace one link with another (replaced link is deleted, children are updated
88         ↪ or deleted), it is actually merge operation
89         return this.Merge(updatedLink, existedLink);
90     }
91 }
92
93 /// <summary>Удаляет связь с указанным индексом.</summary>
94 /// <param name="link">Индекс удаляемой связи.</param>
95 public override void Delete(ulong link)
96 {
97     this.EnsureLinkExists(link);
98     Links.Update(link, Constants.Null, Constants.Null);
99     var referencesCount = Links.Count(Constants.Any, link);
100     if (referencesCount > 0)
101     {
102         var references = new ulong[referencesCount];
103         var referencesFiller = new ArrayFiller<ulong, ulong>(references,
104             ↪ Constants.Continue);
105         Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, link);
106         //references.Sort(); // TODO: Решить необходимо ли для корректного порядка
107         ↪ отмены операций в транзакциях
108         for (var i = (long)referencesCount - 1; i >= 0; i--)
109         {
110             if (this.Exists(references[i]))
111             {
112                 Delete(references[i]);
113             }
114             //else
115             // TODO: Определить почему здесь есть связи, которых не существует
116         }
117     }
118     Links.Delete(link);
119 }
120 }
121 }
122 }

```

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

```

```

77         return false;
78         if (!Equals(matchDecision, Constants.Skip))
79             transitions.Add(new Transition(matchedLink, newValue));
80         return true;
81     };
82     if (!Memory.Each(handler, restriction))
83         return Constants.Break;
84     }
85 }
86 else
87 {
88     Func<T, bool> handler = link =>
89     {
90         var matchedLink = Memory.GetLinkValue(link);
91         var matchDecision = matchedHandler(matchedLink, matchedLink);
92         return !Equals(matchDecision, Constants.Break);
93     };
94     if (!Memory.Each(handler, restriction))
95         return Constants.Break;
96 }
97 }
98 else
99 {
100     if (substitution != null)
101     {
102         transitions = new List<IList<T>>();
103         Func<T, bool> handler = link =>
104         {
105             var matchedLink = Memory.GetLinkValue(link);
106             transitions.Add(matchedLink);
107             return true;
108         };
109         if (!Memory.Each(handler, restriction))
110             return Constants.Break;
111     }
112     else
113     {
114         return Constants.Continue;
115     }
116 }
117 }
118 if (substitution != null)
119 {
120     // Есть причина делать замену (запись)
121     if (substitutedHandler != null)
122     {
123     }
124     else
125     {
126     }
127 }
128 return Constants.Continue;
129
130 //if (restriction.IsNullOrEmpty()) // Create
131 //{
132 //    substitution[Constants.IndexPart] = Memory.AllocateLink();
133 //    Memory.SetLinkValue(substitution);
134 //}
135 //else if (substitution.IsNullOrEmpty()) // Delete
136 //{
137 //    Memory.FreeLink(restriction[Constants.IndexPart]);
138 //}
139 //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
140 //{
141 //    // No need to collect links to list
142 //    // Skip == Continue
143 //    // No need to check substitutedHandler
144 //    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
145 //        Constants.Break), restriction))
146 //        return Constants.Break;
147 //}
148 //else // Update
149 //{
150 //    //List<IList<T>> matchedLinks = null;
151 //    if (matchedHandler != null)
152 //    {
153         matchedLinks = new List<IList<T>>();
154         Func<T, bool> handler = link =>

```

```

154         {
155             var matchedLink = Memory.GetLinkValue(link);
156             var matchDecision = matchedHandler(matchedLink);
157             if (Equals(matchDecision, Constants.Break))
158                 return false;
159             if (!Equals(matchDecision, Constants.Skip))
160                 matchedLinks.Add(matchedLink);
161             return true;
162         };
163         if (!Memory.Each(handler, restriction))
164             return Constants.Break;
165     }
166     if (!matchedLinks.IsNullOrEmpty())
167     {
168         var totalMatchedLinks = matchedLinks.Count;
169         for (var i = 0; i < totalMatchedLinks; i++)
170         {
171             var matchedLink = matchedLinks[i];
172             if (substitutedHandler != null)
173             {
174                 var newValue = new List<T>(); // TODO: Prepare value to update here
175                 // TODO: Decide is it actually needed to use Before and After
176                 substitution handling.
177                 var substitutedDecision = substitutedHandler(matchedLink,
178                     newValue);
179                 if (Equals(substitutedDecision, Constants.Break))
180                     return Constants.Break;
181                 if (Equals(substitutedDecision, Constants.Continue))
182                 {
183                     // Actual update here
184                     Memory.SetLinkValue(newValue);
185                 }
186                 if (Equals(substitutedDecision, Constants.Skip))
187                 {
188                     // Cancel the update. TODO: decide use separate Cancel
189                     constant or Skip is enough?
190                 }
191             }
192         }
193     }
194     return Constants.Continue;
195 }
196
197 public TLink Trigger(ICollection<TLink> patternOrCondition, Func<ICollection<TLink>, TLink>
198     matchHandler, ICollection<TLink> substitution, Func<ICollection<TLink>, ICollection<TLink>, TLink>
199     substitutionHandler)
200 {
201     if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
202     {
203         return Constants.Continue;
204     }
205     else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
206         // Check if it is a correct condition
207     {
208         // Or it only applies to trigger without matchHandler.
209         throw new NotImplementedException();
210     }
211     else if (!substitution.IsNullOrEmpty()) // Creation
212     {
213         var before = ArrayPool<TLink>.Empty;
214         // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
215         // (пройти мимо) или пустить (взять)?
216         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
217             Constants.Break))
218         {
219             return Constants.Break;
220         }
221         var after = (ICollection<TLink>)substitution.ToArray();
222         if (_equalityComparer.Equals(after[0], default))
223         {
224             var newLink = Links.Create();
225             after[0] = newLink;
226         }
227         if (substitution.Count == 1)
228         {
229             after = Links.GetLink(substitution[0]);
230         }
231     }
232 }

```

```

224     else if (substitution.Count == 3)
225     {
226         Links.Update(after);
227     }
228     else
229     {
230         throw new NotSupportedException();
231     }
232     if (matchHandler != null)
233     {
234         return substitutionHandler(before, after);
235     }
236     return Constants.Continue;
237 }
238 else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
239 {
240     if (patternOrCondition.Count == 1)
241     {
242         var linkToDelete = patternOrCondition[0];
243         var before = Links.GetLink(linkToDelete);
244         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
245             ↪ Constants.Break))
246         {
247             return Constants.Break;
248         }
249         var after = ArrayPool<TLink>.Empty;
250         Links.Update(linkToDelete, Constants.Null, Constants.Null);
251         Links.Delete(linkToDelete);
252         if (matchHandler != null)
253         {
254             return substitutionHandler(before, after);
255         }
256         return Constants.Continue;
257     }
258     else
259     {
260         throw new NotSupportedException();
261     }
262 }
263 else // Replace / Update
264 {
265     if (patternOrCondition.Count == 1) //-V3125
266     {
267         var linkToUpdate = patternOrCondition[0];
268         var before = Links.GetLink(linkToUpdate);
269         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
270             ↪ Constants.Break))
271         {
272             return Constants.Break;
273         }
274         var after = (IList<TLink>)substitution.ToArray(); //-V3125
275         if (_equalityComparer.Equals(after[0], default))
276         {
277             after[0] = linkToUpdate;
278         }
279         if (substitution.Count == 1)
280         {
281             if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
282             {
283                 after = Links.GetLink(substitution[0]);
284                 Links.Update(linkToUpdate, Constants.Null, Constants.Null);
285                 Links.Delete(linkToUpdate);
286             }
287         }
288         else if (substitution.Count == 3)
289         {
290             Links.Update(after);
291         }
292         else
293         {
294             throw new NotSupportedException();
295         }
296         if (matchHandler != null)
297         {
298             return substitutionHandler(before, after);
299         }
300         return Constants.Continue;
301     }
302 }

```

```

300         else
301         {
302             throw new NotSupportedException();
303         }
304     }
305 }
306
307 /// <remarks>
308 /// IList[IList[IList[T]]]
309 /// |         |         |         |
310 /// |         |         |         |
311 /// |         |         |         |
312 /// |         |         |         |
313 /// |         |         |         |
314 /// |         |         |         |
315 /// |         |         |         |
316 /// |         |         |         |
317 public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
    ↳ substitution)
318 {
319     var changes = new List<IList<IList<TLink>>>();
320     Trigger(condition, AlwaysContinue, substitution, (before, after) =>
321     {
322         var change = new[] { before, after };
323         changes.Add(change);
324         return Constants.Continue;
325     });
326     return changes;
327 }
328
329 private TLink AlwaysContinue(IList<TLink> linkToMatch) => Constants.Continue;
330 }
331 }

```

./DoubletComparer.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Data.Doublets
5  {
6      /// <remarks>
7      /// TODO: Может стоит попробовать ref во всех методах (IRefEqualityComparer)
8      /// 2x faster with comparer
9      /// </remarks>
10     public class DoubletComparer<T> : IEqualityComparer<Doublet<T>>
11     {
12         private static readonly EqualityComparer<T> _equalityComparer =
            ↳ EqualityComparer<T>.Default;
13
14         public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public bool Equals(Doublet<T> x, Doublet<T> y) => _equalityComparer.Equals(x.Source,
            ↳ y.Source) && _equalityComparer.Equals(x.Target, y.Target);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public int GetHashCode(Doublet<T> obj) => unchecked(obj.Source.GetHashCode() << 15 ~
            ↳ obj.Target.GetHashCode());
21     }
22 }

```

./Doublet.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets
5  {
6      public struct Doublet<T> : IEquatable<Doublet<T>>
7      {
8         private static readonly EqualityComparer<T> _equalityComparer =
            ↳ EqualityComparer<T>.Default;
9
10         public T Source { get; set; }
11         public T Target { get; set; }
12
13         public Doublet(T source, T target)
14         {
15             Source = source;
16             Target = target;
17         }
18     }
19 }

```

```

17     }
18
19     public override string ToString() => $"{Source}->{Target}";
20
21     public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
22     ↪ && _equalityComparer.Equals(Target, other.Target);
23 }

```

./Hybrid.cs

```

1  using System;
2  using System.Reflection;
3  using Platform.Reflection;
4  using Platform.Converters;
5  using Platform.Numbers;
6
7  namespace Platform.Data.Doublets
8  {
9      public class Hybrid<T>
10     {
11         public readonly T Value;
12         public bool IsNothing => Convert.ToInt64(To.Signed(Value)) == 0;
13         public bool IsInternal => Convert.ToInt64(To.Signed(Value)) > 0;
14         public bool IsExternal => Convert.ToInt64(To.Signed(Value)) < 0;
15         public long AbsoluteValue => Math.Abs(Convert.ToInt64(To.Signed(Value)));
16
17         public Hybrid(T value)
18         {
19             if (CachedTypeInfo<T>.IsSigned)
20             {
21                 throw new NotSupportedException();
22             }
23             Value = value;
24         }
25
26         public Hybrid(object value) => Value = To.UnsignedAs<T>(Convert.ChangeType(value,
27             ↪ CachedTypeInfo<T>.SignedVersion));
28
29         public Hybrid(object value, bool isExternal)
30         {
31             var signedType = CachedTypeInfo<T>.SignedVersion;
32             var signedValue = Convert.ChangeType(value, signedType);
33             var abs =
34             ↪ typeof(MathHelpers).GetTypeInfo().GetMethod("Abs").MakeGenericMethod(signedType);
35             var negate = typeof(MathHelpers).GetTypeInfo().GetMethod("Negate").MakeGenericMethod
36             ↪ (signedType);
37             var absoluteValue = abs.Invoke(null, new[] { signedValue });
38             var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
39             ↪ absoluteValue;
40             Value = To.UnsignedAs<T>(resultValue);
41         }
42
43         public static implicit operator Hybrid<T>(T integer) => new Hybrid<T>(integer);
44         public static explicit operator Hybrid<T>(ulong integer) => new Hybrid<T>(integer);
45         public static explicit operator Hybrid<T>(long integer) => new Hybrid<T>(integer);
46         public static explicit operator Hybrid<T>(uint integer) => new Hybrid<T>(integer);
47         public static explicit operator Hybrid<T>(int integer) => new Hybrid<T>(integer);
48         public static explicit operator Hybrid<T>(ushort integer) => new Hybrid<T>(integer);
49         public static explicit operator Hybrid<T>(short integer) => new Hybrid<T>(integer);
50         public static explicit operator Hybrid<T>(byte integer) => new Hybrid<T>(integer);
51         public static explicit operator Hybrid<T>(sbyte integer) => new Hybrid<T>(integer);
52         public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;
53         public static explicit operator ulong(Hybrid<T> hybrid) =>
54             ↪ Convert.ToUInt64(hybrid.Value);
55         public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;
56         public static explicit operator uint(Hybrid<T> hybrid) => Convert.ToUInt32(hybrid.Value);
57
58     }
59
60
61
62
63
64

```



```

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

```

./ILinks.cs

```

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

```

./ILinksExtensions.cs

```

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

```

```

46         links.Delete(link);
47         if ((Integer<TLink>)links.Count() < min)
48         {
49             break;
50         }
51     }
52 }
53
54 /// <remarks>
55 /// TODO: Возможно есть очень простой способ это сделать.
56 /// (Например просто удалить файл, или изменить его размер таким образом,
57 /// чтобы удалился весь контент)
58 /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
59 /// </remarks>
60 public static void DeleteAll<TLink>(this ILinks<TLink> links)
61 {
62     var equalityComparer = EqualityComparer<TLink>.Default;
63     var comparer = Comparer<TLink>.Default;
64     for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
        ↪ ArithmeticHelpers.Decrement(i))
65     {
66         links.Delete(i);
67         if (!equalityComparer.Equals(links.Count(), ArithmeticHelpers.Decrement(i)))
68         {
69             i = links.Count();
70         }
71     }
72 }
73
74 public static TLink First<TLink>(this ILinks<TLink> links)
75 {
76     TLink firstLink = default;
77     var equalityComparer = EqualityComparer<TLink>.Default;
78     if (equalityComparer.Equals(links.Count(), default))
79     {
80         throw new Exception("В хранилище нет связей.");
81     }
82     links.Each(links.Constants.Any, links.Constants.Any, link =>
83     {
84         firstLink = link[links.Constants.IndexPart];
85         return links.Constants.Break;
86     });
87     if (equalityComparer.Equals(firstLink, default))
88     {
89         throw new Exception("В процессе поиска по хранилищу не было найдено связей.");
90     }
91     return firstLink;
92 }
93
94 public static bool IsInnerReference<TLink>(this ILinks<TLink> links, TLink reference)
95 {
96     var constants = links.Constants;
97     var comparer = Comparer<TLink>.Default;
98     return comparer.Compare(constants.MinPossibleIndex, reference) >= 0 &&
        ↪ comparer.Compare(reference, constants.MaxPossibleIndex) <= 0;
99 }
100
101 #region Paths
102
103 /// <remarks>
104 /// TODO: Как так? Как то что ниже может быть корректно?
105 /// Скорее всего практически не применимо
106 /// Предполагалось, что можно было конвертировать формируемый в проходе через
        ↪ SequenceWalker
107 /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
108 /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
109 /// </remarks>
110 public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
        ↪ path)
111 {
112     var current = path[0];
113     //EnsureLinkExists(current, "path");
114     if (!links.Exists(current))
115     {
116         return false;
117     }
118     var equalityComparer = EqualityComparer<TLink>.Default;
119     var constants = links.Constants;
120     for (var i = 1; i < path.Length; i++)

```

```

121     {
122         var next = path[i];
123         var values = links.GetLink(current);
124         var source = values[constants.SourcePart];
125         var target = values[constants.TargetPart];
126         if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
127             ↪ next))
128         {
129             //throw new Exception(string.Format("Невозможно выбрать путь, так как и
130             ↪ Source и Target совпадают с элементом пути {0}.", next));
131             return false;
132         }
133         if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
134             ↪ target))
135         {
136             //throw new Exception(string.Format("Невозможно продолжить путь через
137             ↪ элемент пути {0}", next));
138             return false;
139         }
140         current = next;
141     }
142     return true;
143 }
144
145 /// <remarks>
146 /// Может потребовать дополнительного стека для PathElement's при использовании
147 ↪ SequenceWalker.
148 /// </remarks>
149 public static TLink GetByKeyes<TLink>(this ILinks<TLink> links, TLink root, params int[]
150     ↪ path)
151 {
152     links.EnsureLinkExists(root, "root");
153     var currentLink = root;
154     for (var i = 0; i < path.Length; i++)
155     {
156         currentLink = links.GetLink(currentLink)[path[i]];
157     }
158     return currentLink;
159 }
160
161 public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
162     ↪ links, TLink root, ulong size, ulong index)
163 {
164     var constants = links.Constants;
165     var source = constants.SourcePart;
166     var target = constants.TargetPart;
167     if (!MathHelpers.IsPowerOfTwo(size))
168     {
169         throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
170             ↪ than powers of two are not supported.");
171     }
172     var path = new BitArray(BitConverter.GetBytes(index));
173     var length = BitwiseHelpers.GetLowestBitPosition(size);
174     links.EnsureLinkExists(root, "root");
175     var currentLink = root;
176     for (var i = length - 1; i >= 0; i--)
177     {
178         currentLink = links.GetLink(currentLink)[path[i] ? target : source];
179     }
180     return currentLink;
181 }
182
183 #endregion
184
185 /// <summary>
186 /// Возвращает индекс указанной связи.
187 /// </summary>
188 /// <param name="links">Хранилище связей.</param>
189 /// <param name="link">Связь представленная списком, состоящим из её адреса и
190     ↪ содержимого.</param>
191 /// <returns>Индекс начальной связи для указанной связи.</returns>
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
194     ↪ link[links.Constants.IndexPart];
195
196 /// <summary>
197 /// Возвращает индекс начальной (Source) связи для указанной связи.
198 /// </summary>

```

```

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

```

```

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

```

```

309 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↳ restrictions)
310 {
311     for (int i = 0; i < restrictions.Count; i++)
312     {
313         links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
314     }
315 }
316
317 [MethodImpl(MethodImplOptions.AggressiveInlining)]
318 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
    ↳ string argumentName)
319 {
320     var equalityComparer = EqualityComparer<TLink>.Default;
321     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
322     {
323         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
324     }
325 }
326
327 [MethodImpl(MethodImplOptions.AggressiveInlining)]
328 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
    ↳ link, string argumentName)
329 {
330     var equalityComparer = EqualityComparer<TLink>.Default;
331     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
332     {
333         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
334     }
335 }
336
337 /// <param name="links">Хранилище связей.</param>
338 [MethodImpl(MethodImplOptions.AggressiveInlining)]
339 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target)
340 {
341     if (links.Exists(source, target))
342     {
343         throw new LinkWithSameValueAlreadyExistsException();
344     }
345 }
346
347 /// <param name="links">Хранилище связей.</param>
348 public static void EnsureNoDependencies<TLink>(this ILinks<TLink> links, TLink link)
349 {
350     if (links.DependenciesExist(link))
351     {
352         throw new ArgumentLinkHasDependenciesException<TLink>(link);
353     }
354 }
355
356 /// <param name="links">Хранилище связей.</param>
357 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.Create, addresses);
358
359 /// <param name="links">Хранилище связей.</param>
360 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
361
362 /// <param name="links">Хранилище связей.</param>
363 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
    ↳ params TLink[] addresses)
364 {
365     var constants = links.Constants;
366     var nonExistentAddresses = new HashSet<ulong>(addresses.Where(x =>
    ↳ !links.Exists(x)).Select(x => (ulong)(Integer<TLink>)x));
367     if (nonExistentAddresses.Count > 0)
368     {
369         var max = nonExistentAddresses.Max();
370         // TODO: Эту верхнюю границу нужно разрешить переопределять (проверить
    ↳ применяется ли эта логика)
371         max = Math.Min(max, (Integer<TLink>)constants.MaxPossibleIndex);
372         var createdLinks = new List<TLink>();
373         var equalityComparer = EqualityComparer<TLink>.Default;
374         TLink createdLink = creator();
375         while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
376         {
377             createdLinks.Add(createdLink);

```

```

378     }
379     for (var i = 0; i < createdLinks.Count; i++)
380     {
381         if (!nonExistentAddresses.Contains((Integer<TLink>)createdLinks[i]))
382         {
383             links.Delete(createdLinks[i]);
384         }
385     }
386 }
387
388 #endregion
389
390 /// <param name="links">Хранилище связей.</param>
391 public static ulong DependenciesCount<TLink>(this ILinks<TLink> links, TLink link)
392 {
393     var constants = links.Constants;
394     var values = links.GetLink(link);
395     ulong referencesAsSource = (Integer<TLink>)links.Count(constants.Any, link,
396         ↪ constants.Any);
397     var equalityComparer = EqualityComparer<TLink>.Default;
398     if (equalityComparer.Equals(values[constants.SourcePart], link))
399     {
400         referencesAsSource--;
401     }
402     ulong referencesAsTarget = (Integer<TLink>)links.Count(constants.Any, constants.Any,
403         ↪ link);
404     if (equalityComparer.Equals(values[constants.TargetPart], link))
405     {
406         referencesAsTarget--;
407     }
408     return referencesAsSource + referencesAsTarget;
409 }
410
411 /// <param name="links">Хранилище связей.</param>
412 [MethodImpl(MethodImplOptions.AggressiveInlining)]
413 public static bool DependenciesExist<TLink>(this ILinks<TLink> links, TLink link) =>
414     ↪ links.DependenciesCount(link) > 0;
415
416 /// <param name="links">Хранилище связей.</param>
417 [MethodImpl(MethodImplOptions.AggressiveInlining)]
418 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
419     ↪ TLink target)
420 {
421     var constants = links.Constants;
422     var values = links.GetLink(link);
423     var equalityComparer = EqualityComparer<TLink>.Default;
424     return equalityComparer.Equals(values[constants.SourcePart], source) &&
425         ↪ equalityComparer.Equals(values[constants.TargetPart], target);
426 }
427
428 /// <summary>
429 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
430 /// </summary>
431 /// <param name="links">Хранилище связей.</param>
432 /// <param name="source">Индекс связи, которая является началом для искомой
433     ↪ связи.</param>
434 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
435 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
436     ↪ (концом).</returns>
437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
438 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
439     ↪ target)
440 {
441     var constants = links.Constants;
442     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
443     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
444     return setter.Result;
445 }
446
447 /// <param name="links">Хранилище связей.</param>
448 [MethodImpl(MethodImplOptions.AggressiveInlining)]
449 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
450 {
451     var link = links.Create();
452     return links.Update(link, link, link);
453 }

```

```

448 /// <param name="links">Хранилище связей.</param>
449 [MethodImpl(MethodImplOptions.AggressiveInlining)]
450 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target) => links.Update(links.Create(), source, target);
451
452 /// <summary>
453 /// Обновляет связь с указанными началом (Source) и концом (Target)
454 /// на связь с указанными началом (NewSource) и концом (NewTarget).
455 /// </summary>
456 /// <param name="links">Хранилище связей.</param>
457 /// <param name="link">Индекс обновляемой связи.</param>
458 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↳ выполняется обновление.</param>
459 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↳ выполняется обновление.</param>
460 /// <returns>Индекс обновлённой связи.</returns>
461 [MethodImpl(MethodImplOptions.AggressiveInlining)]
462 public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
    ↳ TLink newTarget) => links.Update(new[] { link, newSource, newTarget });
463
464 /// <summary>
465 /// Обновляет связь с указанными началом (Source) и концом (Target)
466 /// на связь с указанными началом (NewSource) и концом (NewTarget).
467 /// </summary>
468 /// <param name="links">Хранилище связей.</param>
469 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
    ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
    ↳ Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
    ↳ связи.</param>
470 /// <returns>Индекс обновлённой связи.</returns>
471 [MethodImpl(MethodImplOptions.AggressiveInlining)]
472 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
473 {
474     if (restrictions.Length == 2)
475     {
476         return links.Merge(restrictions[0], restrictions[1]);
477     }
478     if (restrictions.Length == 4)
479     {
480         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
            ↳ restrictions[2], restrictions[3]);
481     }
482     else
483     {
484         return links.Update(restrictions);
485     }
486 }
487
488 /// <summary>
489 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
    ↳ с указанными Source (началом) и Target (концом).
490 /// </summary>
491 /// <param name="links">Хранилище связей.</param>
492 /// <param name="source">Индекс связи, которая является началом на создаваемой
    ↳ связи.</param>
493 /// <param name="target">Индекс связи, которая является концом для создаваемой
    ↳ связи.</param>
494 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)
497 {
498     var link = links.SearchOrDefault(source, target);
499     if (EqualityComparer<TLink>.Default.Equals(link, default))
500     {
501         link = links.CreateAndUpdate(source, target);
502     }
503     return link;
504 }
505
506 /// <summary>
507 /// Обновляет связь с указанными началом (Source) и концом (Target)
508 /// на связь с указанными началом (NewSource) и концом (NewTarget).
509 /// </summary>
510 /// <param name="links">Хранилище связей.</param>
511 /// <param name="source">Индекс связи, которая является началом обновляемой
    ↳ связи.</param>

```



```

512 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
513 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↳ выполняется обновление.</param>
514 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↳ выполняется обновление.</param>
515 /// <returns>Индекс обновлённой связи.</returns>
516 [MethodImpl(MethodImplOptions.AggressiveInlining)]
517 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target, TLink newSource, TLink newTarget)
518 {
519     var equalityComparer = EqualityComparer<TLink>.Default;
520     var link = links.SearchOrDefault(source, target);
521     if (equalityComparer.Equals(link, default))
522     {
523         return links.CreateAndUpdate(newSource, newTarget);
524     }
525     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
    ↳ target))
526     {
527         return link;
528     }
529     return links.Update(link, newSource, newTarget);
530 }
531
532 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
533 /// <param name="links">Хранилище связей.</param>
534 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
535 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
536 [MethodImpl(MethodImplOptions.AggressiveInlining)]
537 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)
538 {
539     var link = links.SearchOrDefault(source, target);
540     if (!EqualityComparer<TLink>.Default.Equals(link, default))
541     {
542         links.Delete(link);
543         return link;
544     }
545     return default;
546 }
547
548 /// <summary>Удаляет несколько связей.</summary>
549 /// <param name="links">Хранилище связей.</param>
550 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
551 [MethodImpl(MethodImplOptions.AggressiveInlining)]
552 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
553 {
554     for (int i = 0; i < deletedLinks.Count; i++)
555     {
556         links.Delete(deletedLinks[i]);
557     }
558 }
559
560 // Replace one link with another (replaced link is deleted, children are updated or
    ↳ deleted)
561 public static TLink Merge<TLink>(this ILinks<TLink> links, TLink linkIndex, TLink
    ↳ newLink)
562 {
563     var equalityComparer = EqualityComparer<TLink>.Default;
564     if (equalityComparer.Equals(linkIndex, newLink))
565     {
566         return newLink;
567     }
568     var constants = links.Constants;
569     ulong referencesAsSourceCount = (Integer<TLink>)links.Count(constants.Any,
    ↳ linkIndex, constants.Any);
570     ulong referencesAsTargetCount = (Integer<TLink>)links.Count(constants.Any,
    ↳ constants.Any, linkIndex);
571     var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(linkIndex)) &&
    ↳ referencesAsSourceCount == 1 && referencesAsTargetCount == 1;
572     if (!isStandalonePoint)
573     {
574         var totalReferences = referencesAsSourceCount + referencesAsTargetCount;
575         if (totalReferences > 0)
576         {
577             var references = ArrayPool.Allocate<TLink>((long)totalReferences);
578             var referencesFiller = new ArrayFiller<TLink, TLink>(references,
    ↳ links.Constants.Continue);

```

```

579         links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any,
580             ↪ linkIndex, constants.Any);
581         links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any,
582             ↪ constants.Any, linkIndex);
583         for (ulong i = 0; i < referencesAsSourceCount; i++)
584         {
585             var reference = references[i];
586             if (equalityComparer.Equals(reference, linkIndex))
587             {
588                 continue;
589             }
590             links.Update(reference, newLink, links.GetTarget(reference));
591         }
592         for (var i = (long)referencesAsSourceCount; i < references.Length; i++)
593         {
594             var reference = references[i];
595             if (equalityComparer.Equals(reference, linkIndex))
596             {
597                 continue;
598             }
599             links.Update(reference, links.GetSource(reference), newLink);
600         }
601         ArrayPool.Free(references);
602     }
603 }
604 links.Delete(linkIndex);
605 return newLink;
606 }
607 }
608 }

```

./Incrementers/FrequencyIncrementer.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Incrementers
5  {
6      public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
7      {
8          private static readonly EqualityComparer<TLink> _equalityComparer =
9              ↪ EqualityComparer<TLink>.Default;
10
11          private readonly TLink _frequencyMarker;
12          private readonly TLink _unaryOne;
13          private readonly IIncrementer<TLink> _unaryNumberIncrementer;
14
15          public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
16              ↪ IIncrementer<TLink> unaryNumberIncrementer)
17              : base(links)
18          {
19              _frequencyMarker = frequencyMarker;
20              _unaryOne = unaryOne;
21              _unaryNumberIncrementer = unaryNumberIncrementer;
22          }
23
24          public TLink Increment(TLink frequency)
25          {
26              if (_equalityComparer.Equals(frequency, default))
27              {
28                  return Links.GetOrCreate(_unaryOne, _frequencyMarker);
29              }
30              var source = Links.GetSource(frequency);
31              var incrementedSource = _unaryNumberIncrementer.Increment(source);
32              return Links.GetOrCreate(incrementedSource, _frequencyMarker);
33          }
34      }
35  }

```

./Incrementers/LinkFrequencyIncrementer.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Incrementers
5  {
6      public class LinkFrequencyIncrementer<TLink> : LinksOperatorBase<TLink>,
7          ↪ IIncrementer<IList<TLink>>
8      {

```

```

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

```

./Incrementers/UnaryNumberIncrementer.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Incrementers
5 {
6     public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
7     {
8         private static readonly EqualityComparer<TLink> _equalityComparer =
9         ↪ EqualityComparer<TLink>.Default;
10
11         private readonly TLink _unaryOne;
12
13         public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
14         ↪ _unaryOne = unaryOne;
15
16         public TLink Increment(TLink unaryNumber)
17         {
18             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
19             {
20                 return Links.GetOrCreate(_unaryOne, _unaryOne);
21             }
22             var source = Links.GetSource(unaryNumber);
23             var target = Links.GetTarget(unaryNumber);
24             if (_equalityComparer.Equals(source, target))
25             {
26                 return Links.GetOrCreate(unaryNumber, _unaryOne);
27             }
28             else
29             {
30                 return Links.GetOrCreate(source, Increment(target));
31             }
32         }
33     }
34 }

```

./ISynchronizedLinks.cs

```

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

```

```

./Link.cs
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Platform.Exceptions;
5  using Platform.Ranges;
6  using Platform.Helpers.Singletons;
7  using Platform.Data.Constants;
8
9  namespace Platform.Data.Doublets
10 {
11     /// <summary>
12     /// Структура описывающая уникальную связь.
13     /// </summary>
14     public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
15     {
16         public static readonly Link<TLink> Null = new Link<TLink>();
17
18         private static readonly LinksCombinedConstants<bool, TLink, int> _constants =
19             ↪ Default<LinksCombinedConstants<bool, TLink, int>>.Instance;
20         private static readonly EqualityComparer<TLink> _equalityComparer =
21             ↪ EqualityComparer<TLink>.Default;
22
23         private const int Length = 3;
24
25         public readonly TLink Index;
26         public readonly TLink Source;
27         public readonly TLink Target;
28
29         public Link(params TLink[] values)
30         {
31             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
32                 ↪ _constants.Null;
33             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
34                 ↪ _constants.Null;
35             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
36                 ↪ _constants.Null;
37         }
38
39         public Link(IList<TLink> values)
40         {
41             Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
42                 ↪ _constants.Null;
43             Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
44                 ↪ _constants.Null;
45             Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
46                 ↪ _constants.Null;
47         }
48
49         public Link(TLink index, TLink source, TLink target)
50         {
51             Index = index;
52             Source = source;
53             Target = target;
54         }
55
56         public Link(TLink source, TLink target)
57             : this(_constants.Null, source, target)
58         {
59             Source = source;
60             Target = target;
61         }
62
63         public static Link<TLink> Create(TLink source, TLink target) => new Link<TLink>(source,
64             ↪ target);
65
66         public override int GetHashCode() => (Index, Source, Target).GetHashCode();
67
68         public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
69             && _equalityComparer.Equals(Source, _constants.Null)
70             && _equalityComparer.Equals(Target, _constants.Null);
71
72         public override bool Equals(object other) => other is Link<TLink> &&
73             ↪ Equals((Link<TLink>)other);
74
75         public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
76             && _equalityComparer.Equals(Source, other.Source)
77             && _equalityComparer.Equals(Target, other.Target);

```

```

69     public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
70         ↳ {source}->{target})";
71
72     public static string ToString(TLink source, TLink target) => $"({source}->{target})";
73
74     public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
75
76     public static implicit operator Link<TLink>(TLink[] linkArray) => new
77         ↳ Link<TLink>(linkArray);
78
79     public TLink[] ToArray()
80     {
81         var array = new TLink[Length];
82         CopyTo(array, 0);
83         return array;
84     }
85
86     public override string ToString() => _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, Length - 1),
100                 ↳ nameof(index));
101             if (index == _constants.IndexPart)
102             {
103                 return Index;
104             }
105             if (index == _constants.SourcePart)
106             {
107                 return Source;
108             }
109             if (index == _constants.TargetPart)
110             {
111                 return Target;
112             }
113             throw new NotSupportedException(); // Impossible path due to
114                 ↳ 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 NotSupportedException();
129
130     public void Clear() => throw new NotSupportedException();
131
132     public bool Contains(TLink item) => IndexOf(item) >= 0;
133
134     public void CopyTo(TLink[] array, int arrayIndex)
135     {
136         Ensure.Always.ArgumentNotNull(array, nameof(array));
137         Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
138             ↳ nameof(arrayIndex));
139         if (arrayIndex + Length > array.Length)
140         {
141             throw new InvalidOperationException();
142         }
143         array[arrayIndex++] = Index;
144         array[arrayIndex++] = Source;
145         array[arrayIndex] = Target;
146     }

```

```

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

```

./LinkExtensions.cs

```

1 namespace Platform.Data.Doublets
2 {
3     public static class LinkExtensions
4     {
5         public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
6             ↪ Point<TLink>.IsFullPoint(link);
7         public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
8             ↪ Point<TLink>.IsPartialPoint(link);
9     }
10 }

```

./LinksOperatorBase.cs

```

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

```

./PropertyOperators/DefaultLinkPropertyOperator.cs

```

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

```

```

28         return value;
29     }
30
31     public void SetValue(TLink @object, TLink property, TLink value)
32     {
33         var objectProperty = Links.GetOrCreate(@object, property);
34         Links.DeleteMany(Links.All(Links.Constants.Any, objectProperty).Select(link =>
35             ↪ link[Links.Constants.IndexPart]).ToList());
36         Links.GetOrCreate(objectProperty, value);
37     }
38 }

```

./PropertyOperators/FrequencyPropertyOperator.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.PropertyOperators
5  {
6      public class FrequencyPropertyOperator<TLink> : LinksOperatorBase<TLink>,
7          ↪ ISpecificPropertyOperator<TLink, TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↪ EqualityComparer<TLink>.Default;
11
12         private readonly TLink _frequencyPropertyMarker;
13         private readonly TLink _frequencyMarker;
14
15         public FrequencyPropertyOperator(ILinks<TLink> links, TLink frequencyPropertyMarker,
16             ↪ TLink frequencyMarker) : base(links)
17         {
18             _frequencyPropertyMarker = frequencyPropertyMarker;
19             _frequencyMarker = frequencyMarker;
20         }
21
22         public TLink Get(TLink link)
23         {
24             var property = Links.SearchOrDefault(link, _frequencyPropertyMarker);
25             var container = GetContainer(property);
26             var frequency = GetFrequency(container);
27             return frequency;
28         }
29
30         private TLink GetContainer(TLink property)
31         {
32             var frequencyContainer = default(TLink);
33             if (_equalityComparer.Equals(property, default))
34             {
35                 return frequencyContainer;
36             }
37             Links.Each(candidate =>
38             {
39                 var candidateTarget = Links.GetTarget(candidate);
40                 var frequencyTarget = Links.GetTarget(candidateTarget);
41                 if (_equalityComparer.Equals(frequencyTarget, _frequencyMarker))
42                 {
43                     frequencyContainer = Links.GetIndex(candidate);
44                     return Links.Constants.Break;
45                 }
46                 return Links.Constants.Continue;
47             }, Links.Constants.Any, property, Links.Constants.Any);
48             return frequencyContainer;
49         }
50
51         private TLink GetFrequency(TLink container) => _equalityComparer.Equals(container,
52             ↪ default) ? default : Links.GetTarget(container);
53
54         public void Set(TLink link, TLink frequency)
55         {
56             var property = Links.GetOrCreate(link, _frequencyPropertyMarker);
57             var container = GetContainer(property);
58             if (_equalityComparer.Equals(container, default))
59             {
60                 Links.GetOrCreate(property, frequency);
61             }
62             else
63             {
64                 Links.Update(container, property, frequency);
65             }
66         }
67     }
68 }

```

```

62     }
63 }
64 }

```

./ResizableDirectMemory/ResizableDirectMemoryLinks.cs

```

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

```



```

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

```

```

117     public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
118         ↪ FreeLinksOffset).GetValue<TLink>();
119     [MethodImpl(MethodImplOptions.AggressiveInlining)]
120     public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
121         ↪ FirstFreeLinkOffset).GetValue<TLink>();
122     [MethodImpl(MethodImplOptions.AggressiveInlining)]
123     public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
124         ↪ FirstAsSourceOffset).GetValue<TLink>();
125     [MethodImpl(MethodImplOptions.AggressiveInlining)]
126     public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
127         ↪ FirstAsTargetOffset).GetValue<TLink>();
128     [MethodImpl(MethodImplOptions.AggressiveInlining)]
129     public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) => pointer +
130         ↪ FirstAsSourceOffset;
131     [MethodImpl(MethodImplOptions.AggressiveInlining)]
132     public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
133         ↪ FirstAsTargetOffset;
134     [MethodImpl(MethodImplOptions.AggressiveInlining)]
135     public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
136         ↪ AllocatedLinksOffset).SetValue(value);
137     [MethodImpl(MethodImplOptions.AggressiveInlining)]
138     public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
139         ↪ ReservedLinksOffset).SetValue(value);
140     [MethodImpl(MethodImplOptions.AggressiveInlining)]
141     public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
142         ↪ FreeLinksOffset).SetValue(value);
143     [MethodImpl(MethodImplOptions.AggressiveInlining)]
144     public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
145         ↪ FirstFreeLinkOffset).SetValue(value);
146     [MethodImpl(MethodImplOptions.AggressiveInlining)]
147     public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
148         ↪ FirstAsSourceOffset).SetValue(value);
149     [MethodImpl(MethodImplOptions.AggressiveInlining)]
150     public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
151         ↪ FirstAsTargetOffset).SetValue(value);
152     [MethodImpl(MethodImplOptions.AggressiveInlining)]
153     public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
154         ↪ LastFreeLinkOffset).SetValue(value);
155 }
156
157 private readonly long _memoryReservationStep;
158
159 private readonly IResizableDirectMemory _memory;
160 private IntPtr _header;
161 private IntPtr _links;
162
163 private LinksTargetsTreeMethods _targetsTreeMethods;
164 private LinksSourcesTreeMethods _sourcesTreeMethods;
165
166 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
167 ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
168 ↪ наличие связи внутри
169 private UnusedLinksListMethods _unusedLinksListMethods;
170
171 /// <summary>
172 /// Возвращает общее число связей находящихся в хранилище.
173 /// </summary>
174 private TLink Total => Subtract(LinksHeader.GetAllocatedLinks(_header),
175     ↪ LinksHeader.GetFreeLinks(_header));
176
177 public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
178
179 public ResizableDirectMemoryLinks(string address)
180     : this(address, DefaultLinksSizeStep)
181 {
182 }
183
184 /// <summary>
185 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
186 ↪ минимальным шагом расширения базы данных.
187 /// </summary>
188 /// <param name="address">Полный путь к файлу базы данных.</param>

```

```

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

```

```

248         return Integer<TLink>.Zero;
249     }
250 }
251 if (restrictions.Count == 3)
252 {
253     var index = restrictions[Constants.IndexPart];
254     var source = restrictions[Constants.SourcePart];
255     var target = restrictions[Constants.TargetPart];
256
257     if (_equalityComparer.Equals(index, Constants.Any))
258     {
259         if (_equalityComparer.Equals(source, Constants.Any) &&
260             ↪ _equalityComparer.Equals(target, Constants.Any))
261         {
262             return Total;
263         }
264         else if (_equalityComparer.Equals(source, Constants.Any))
265         {
266             return _targetsTreeMethods.CalculateReferences(target);
267         }
268         else if (_equalityComparer.Equals(target, Constants.Any))
269         {
270             return _sourcesTreeMethods.CalculateReferences(source);
271         }
272         else //if(source != Any && target != Any)
273         {
274             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
275             var link = _sourcesTreeMethods.Search(source, target);
276             return _equalityComparer.Equals(link, Constants.Null) ?
277                 ↪ Integer<TLink>.Zero : Integer<TLink>.One;
278         }
279     }
280     else
281     {
282         if (!Exists(index))
283         {
284             return Integer<TLink>.Zero;
285         }
286         if (_equalityComparer.Equals(source, Constants.Any) &&
287             ↪ _equalityComparer.Equals(target, Constants.Any))
288         {
289             return Integer<TLink>.One;
290         }
291         var storedLinkValue = GetLinkUnsafe(index);
292         if (!_equalityComparer.Equals(source, Constants.Any) &&
293             ↪ !_equalityComparer.Equals(target, Constants.Any))
294         {
295             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
296                 ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
297             {
298                 return Integer<TLink>.One;
299             }
300             return Integer<TLink>.Zero;
301         }
302         if (_equalityComparer.Equals(source, Constants.Any))
303         {
304             value = target;
305         }
306         if (_equalityComparer.Equals(target, Constants.Any))
307         {
308             value = source;
309         }
310         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
311             ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
312         {
313             return Integer<TLink>.One;
314         }
315         return Integer<TLink>.Zero;
316     }
317 }
318 throw new NotSupportedException("Другие размеры и способы ограничений не
319     ↪ поддерживаются.");
320 }
321
322 [MethodImpl(MethodImplOptions.AggressiveInlining)]
323 public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
324 {

```

```

321 if (restrictions.Count == 0)
322 {
323     for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
324         ↪ (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
325         ↪ Increment(link))
326     {
327         if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
328             ↪ Constants.Break))
329         {
330             return Constants.Break;
331         }
332     }
333     return Constants.Continue;
334 }
335 if (restrictions.Count == 1)
336 {
337     var index = restrictions[Constants.IndexPart];
338     if (_equalityComparer.Equals(index, Constants.Any))
339     {
340         return Each(handler, ArrayPool<TLink>.Empty);
341     }
342     if (!Exists(index))
343     {
344         return Constants.Continue;
345     }
346     return handler(GetLinkStruct(index));
347 }
348 if (restrictions.Count == 2)
349 {
350     var index = restrictions[Constants.IndexPart];
351     var value = restrictions[1];
352     if (_equalityComparer.Equals(index, Constants.Any))
353     {
354         if (_equalityComparer.Equals(value, Constants.Any))
355         {
356             return Each(handler, ArrayPool<TLink>.Empty);
357         }
358         if (_equalityComparer.Equals(Each(handler, new[] { index, value,
359             ↪ Constants.Any }), Constants.Break))
360         {
361             return Constants.Break;
362         }
363         return Each(handler, new[] { index, Constants.Any, 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(storedLinkValue), value) ||
377             ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
378         {
379             return handler(GetLinkStruct(index));
380         }
381         return Constants.Continue;
382     }
383 }
384 if (restrictions.Count == 3)
385 {
386     var index = restrictions[Constants.IndexPart];
387     var source = restrictions[Constants.SourcePart];
388     var target = restrictions[Constants.TargetPart];
389     if (_equalityComparer.Equals(index, Constants.Any))
390     {
391         if (_equalityComparer.Equals(source, Constants.Any) &&
392             ↪ _equalityComparer.Equals(target, Constants.Any))
393         {
394             return Each(handler, ArrayPool<TLink>.Empty);
395         }
396         else if (_equalityComparer.Equals(source, Constants.Any))
397         {

```

```

394         return _targetsTreeMethods.EachReference(target, handler);
395     }
396     else if (_equalityComparer.Equals(target, Constants.Any))
397     {
398         return _sourcesTreeMethods.EachReference(source, handler);
399     }
400     else //if(source != Any && target != Any)
401     {
402         var link = _sourcesTreeMethods.Search(source, target);
403         return _equalityComparer.Equals(link, Constants.Null) ?
            ↳ Constants.Continue : handler(GetLinkStruct(link));
404     }
405 }
406 else
407 {
408     if (!Exists(index))
409     {
410         return Constants.Continue;
411     }
412     if (_equalityComparer.Equals(source, Constants.Any) &&
        ↳ _equalityComparer.Equals(target, Constants.Any))
413     {
414         return handler(GetLinkStruct(index));
415     }
416     var storedLinkValue = GetLinkUnsafe(index);
417     if (!_equalityComparer.Equals(source, Constants.Any) &&
        ↳ !_equalityComparer.Equals(target, Constants.Any))
418     {
419         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
            ↳ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
420         {
421             return handler(GetLinkStruct(index));
422         }
423         return Constants.Continue;
424     }
425     var value = default(TLink);
426     if (_equalityComparer.Equals(source, Constants.Any))
427     {
428         value = target;
429     }
430     if (_equalityComparer.Equals(target, Constants.Any))
431     {
432         value = source;
433     }
434     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
        ↳ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
435     {
436         return handler(GetLinkStruct(index));
437     }
438     return Constants.Continue;
439 }
440 }
441 }
442 throw new NotSupportedException("Другие размеры и способы ограничений не
443 ↳ поддерживаются.");
444 }
445
446 /// <remarks>
447 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
448 ↳ в другом месте (но не в менеджере памяти, а в логике Links)
449 /// </remarks>
450 [MethodImpl(MethodImplOptions.AggressiveInlining)]
451 public TLink Update(IList<TLink> values)
452 {
453     var linkIndex = values[Constants.IndexPart];
454     var link = GetLinkUnsafe(linkIndex);
455     // Будет корректно работать только в том случае, если пространство выделенной связи
456     ↳ предварительно заполнено нулями
457     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
458     {
459         _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
            ↳ linkIndex);
460     }
461     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
462     {
463         _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
            ↳ linkIndex);
464     }
465     Link.SetSource(link, values[Constants.SourcePart]);

```

```

464     Link.SetTarget(link, values[Constants.TargetPart]);
465     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
466     {
467         _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer(_header),
468             ↪ linkIndex);
469     }
470     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
471     {
472         _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTargetPointer(_header),
473             ↪ linkIndex);
474     }
475     return linkIndex;
476 }
477
478 [MethodImpl(MethodImplOptions.AggressiveInlining)]
479 public Link<TLink> GetLinkStruct(TLink linkIndex)
480 {
481     var link = GetLinkUnsafe(linkIndex);
482     return new Link<TLink>(linkIndex, Link.GetSource(link), Link.GetTarget(link));
483 }
484
485 [MethodImpl(MethodImplOptions.AggressiveInlining)]
486 private IntPtr GetLinkUnsafe(TLink linkIndex) => _links.GetElement(LinkSizeInBytes,
487     ↪ linkIndex);
488
489 /// <remarks>
490 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
491 ↪ пространство
492 /// </remarks>
493 public TLink Create()
494 {
495     var freeLink = LinksHeader.GetFirstFreeLink(_header);
496     if (!_equalityComparer.Equals(freeLink, Constants.Null))
497     {
498         _unusedLinksListMethods.Detach(freeLink);
499     }
500     else
501     {
502         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
503             ↪ Constants.MaxPossibleIndex) > 0)
504         {
505             throw new
506                 ↪ LinksLimitReachedException((Integer<TLink>)Constants.MaxPossibleIndex);
507         }
508         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
509             ↪ Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
510         {
511             _memory.ReservedCapacity += _memory.ReservationStep;
512             SetPointers(_memory);
513             LinksHeader.SetReservedLinks(_header,
514                 ↪ (Integer<TLink>)(_memory.ReservedCapacity / LinkSizeInBytes));
515         }
516         LinksHeader.SetAllocatedLinks(_header,
517             ↪ Increment(LinksHeader.GetAllocatedLinks(_header)));
518         _memory.UsedCapacity += LinkSizeInBytes;
519         freeLink = LinksHeader.GetAllocatedLinks(_header);
520     }
521     return freeLink;
522 }
523
524 public void Delete(TLink link)
525 {
526     if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)
527     {
528         _unusedLinksListMethods.AttachAsFirst(link);
529     }
530     else if (_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
531     {
532         LinksHeader.SetAllocatedLinks(_header,
533             ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
534         _memory.UsedCapacity -= LinkSizeInBytes;
535         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
536         ↪ пока не дойдём до первой существующей связи
537         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
538         while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
539             ↪ Integer<TLink>.Zero) > 0) &&
540             ↪ IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
541         {

```

```

529         _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
530         LinksHeader.SetAllocatedLinks(_header,
531         ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
532         _memory.UsedCapacity -= LinkSizeInBytes;
533     }
534 }
535
536 /// <remarks>
537 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
538 ↪ адрес реально поменялся
539 ///
540 /// Указатель this.links может быть в том же месте,
541 /// так как 0-я связь не используется и имеет такой же размер как Header,
542 /// поэтому header размещается в том же месте, что и 0-я связь
543 /// </remarks>
544 private void SetPointers(IDirectMemory memory)
545 {
546     if (memory == null)
547     {
548         _links = IntPtr.Zero;
549         _header = _links;
550         _unusedLinksListMethods = null;
551         _targetsTreeMethods = null;
552         _unusedLinksListMethods = null;
553     }
554     else
555     {
556         _links = memory.Pointer;
557         _header = _links;
558         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
559         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
560         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
561     }
562 }
563
564 [MethodImpl(MethodImplOptions.AggressiveInlining)]
565 private bool Exists(TLink link)
566 => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
567    && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
568    && !IsUnusedLink(link);
569
570 [MethodImpl(MethodImplOptions.AggressiveInlining)]
571 private bool IsUnusedLink(TLink link)
572 => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
573    || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)),
574    ↪ Constants.Null)
575    && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)), Constants.Null));
576
577 #region DisposableBase
578
579 protected override bool AllowMultipleDisposeCalls => true;
580
581 protected override void DisposeCore(bool manual, bool wasDisposed)
582 {
583     if (!wasDisposed)
584     {
585         SetPointers(null);
586     }
587     Disposable.TryDispose(_memory);
588 }
589
590 #endregion
591 }

```

./ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs

```

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

```



```

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

```

./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Numbers;
6  using Platform.Unsafe;
7  using Platform.Collections.Methods.Trees;
8  using Platform.Data.Constants;
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory
11 {
12     partial class ResizableDirectMemoryLinks<TLink>
13     {
14         private abstract class LinksTreeMethodsBase :
15         ↪ SizedAndThreadedAVLBalancedTreeMethods<TLink>
16         {
17             private readonly ResizableDirectMemoryLinks<TLink> _memory;
18             private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
19             protected readonly IntPtr Links;
20             protected readonly IntPtr Header;
21
22             protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
23             {
24                 Links = memory._links;
25                 Header = memory._header;
26                 _memory = memory;
27                 _constants = memory.Constants;
28             }
29
30             [MethodImpl(MethodImplOptions.AggressiveInlining)]
31             protected abstract TLink GetTreeRoot();
32
33             [MethodImpl(MethodImplOptions.AggressiveInlining)]
34             protected abstract TLink GetBasePartValue(TLink link);
35
36             public TLink this[TLink index]
37             {

```

```

37     get
38     {
39         var root = GetTreeRoot();
40         if (GreaterOrEqualThan(index, GetSize(root)))
41         {
42             return GetZero();
43         }
44         while (!EqualToZero(root))
45         {
46             var left = GetLeftOrDefault(root);
47             var leftSize = GetSizeOrZero(left);
48             if (LessThan(index, leftSize))
49             {
50                 root = left;
51                 continue;
52             }
53             if (IsEquals(index, leftSize))
54             {
55                 return root;
56             }
57             root = GetRightOrDefault(root);
58             index = Subtract(index, Increment(leftSize));
59         }
60         return GetZero(); // TODO: Impossible situation exception (only if tree
        ↳ structure broken)
61     }
62 }
63
64 // TODO: Return indices range instead of references count
65 public TLink CalculateReferences(TLink link)
66 {
67     var root = GetTreeRoot();
68     var total = GetSize(root);
69     var totalRightIgnore = GetZero();
70     while (!EqualToZero(root))
71     {
72         var @base = GetBasePartValue(root);
73         if (LessOrEqualThan(@base, link))
74         {
75             root = GetRightOrDefault(root);
76         }
77         else
78         {
79             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
80             root = GetLeftOrDefault(root);
81         }
82     }
83     root = GetTreeRoot();
84     var totalLeftIgnore = GetZero();
85     while (!EqualToZero(root))
86     {
87         var @base = GetBasePartValue(root);
88         if (GreaterOrEqualThan(@base, link))
89         {
90             root = GetLeftOrDefault(root);
91         }
92         else
93         {
94             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
95             root = GetRightOrDefault(root);
96         }
97     }
98     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
99 }
100
101
102 public TLink EachReference(TLink link, Func<IList<TLink>, TLink> handler)
103 {
104     var root = GetTreeRoot();
105     if (EqualToZero(root))
106     {
107         return _constants.Continue;
108     }
109     TLink first = GetZero(), current = root;
110     while (!EqualToZero(current))
111     {
112         var @base = GetBasePartValue(current);
113         if (GreaterOrEqualThan(@base, link))

```

```

114         {
115             if (IsEquals(@base, link))
116             {
117                 first = current;
118             }
119             current = GetLeftOrDefault(current);
120         }
121         else
122         {
123             current = GetRightOrDefault(current);
124         }
125     }
126     if (!EqualToZero(first))
127     {
128         current = first;
129         while (true)
130         {
131             if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
132             {
133                 return _constants.Break;
134             }
135             current = GetNext(current);
136             if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
137             {
138                 break;
139             }
140         }
141     }
142     return _constants.Continue;
143 }
144
145 protected override void PrintNodeValue(TLink node, StringBuilder sb)
146 {
147     sb.Append(' ');
148     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
149         ↪ Link.SourceOffset).GetValue<TLink>());
150     sb.Append('-');
151     sb.Append('>');
152     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
153         ↪ Link.TargetOffset).GetValue<TLink>());
154 }
155
156 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
157 {
158     public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
159         : base(memory)
160     {
161     }
162
163     protected override IntPtr GetLeftPointer(TLink node) =>
164         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset;
165
166     protected override IntPtr GetRightPointer(TLink node) =>
167         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset;
168
169     protected override TLink GetLeftValue(TLink node) =>
170         ↪ (Links.GetElement(LinkSizeInBytes, node) +
171         ↪ Link.LeftAsSourceOffset).GetValue<TLink>();
172
173     protected override TLink GetRightValue(TLink node) =>
174         ↪ (Links.GetElement(LinkSizeInBytes, node) +
175         ↪ Link.RightAsSourceOffset).GetValue<TLink>();
176
177     protected override TLink GetSize(TLink node)
178     {
179         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
180             ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
181         return BitwiseHelpers.PartialRead(previousValue, 5, -5);
182     }
183
184     protected override void SetLeft(TLink node, TLink left) =>
185         ↪ (Links.GetElement(LinkSizeInBytes, node) +
186         ↪ Link.LeftAsSourceOffset).SetValue(left);
187
188     protected override void SetRight(TLink node, TLink right) =>
189         ↪ (Links.GetElement(LinkSizeInBytes, node) +
190         ↪ Link.RightAsSourceOffset).SetValue(right);

```

```

179
180 protected override void SetSize(TLink node, TLink size)
181 {
182     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
183         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
184     (Links.GetElement(LinkSizeInBytes, node) +
185         ↳ Link.SizeAsSourceOffset).SetValue(BitwiseHelpers.PartialWrite(previousValue,
186         ↳ size, 5, -5));
187 }
188
189 protected override bool GetLeftIsChild(TLink node)
190 {
191     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
192         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
193     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 4, 1);
194 }
195
196 protected override void SetLeftIsChild(TLink node, bool value)
197 {
198     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
199         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
200     var modified = BitwiseHelpers.PartialWrite(previousValue,
201         ↳ (TLink)(Integer<TLink>)value, 4, 1);
202     (Links.GetElement(LinkSizeInBytes, node) +
203         ↳ Link.SizeAsSourceOffset).SetValue(modified);
204 }
205
206 protected override bool GetRightIsChild(TLink node)
207 {
208     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
209         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
210     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 3, 1);
211 }
212
213 protected override void SetRightIsChild(TLink node, bool value)
214 {
215     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
216         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
217     var modified = BitwiseHelpers.PartialWrite(previousValue,
218         ↳ (TLink)(Integer<TLink>)value, 3, 1);
219     (Links.GetElement(LinkSizeInBytes, node) +
220         ↳ Link.SizeAsSourceOffset).SetValue(modified);
221 }
222
223 protected override sbyte GetBalance(TLink node)
224 {
225     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
226         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
227     var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 0,
228         ↳ 3);
229     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
230         ↳ 124 : value & 3);
231     return unpackedValue;
232 }
233
234 protected override void SetBalance(TLink node, sbyte value)
235 {
236     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
237         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
238     var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
239         ↳ 3);
240     var modified = BitwiseHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
241     (Links.GetElement(LinkSizeInBytes, node) +
242         ↳ Link.SizeAsSourceOffset).SetValue(modified);
243 }
244
245 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
246 {
247     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
248         ↳ Link.SourceOffset).GetValue<TLink>();
249     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
250         ↳ Link.SourceOffset).GetValue<TLink>();
251     return LessThan(firstSource, secondSource) ||

```

```

233         (IsEquals(firstSource, secondSource) &&
            ↳ LessThan((Links.GetElement(LinkSizeInBytes, first) +
            ↳ Link.TargetOffset).GetValue<TLink>(),
            ↳ (Links.GetElement(LinkSizeInBytes, second) +
            ↳ Link.TargetOffset).GetValue<TLink>()));
234     }
235
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) &&
                ↳ GreaterThan((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)
288     : base(memory)
289 {
290 }
291
292 protected override IntPtr GetLeftPointer(TLink node) =>
293     ↳ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset;
294
295 protected override IntPtr GetRightPointer(TLink node) =>
296     ↳ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset;
297
298 protected override TLink GetLeftValue(TLink node) =>
299     ↳ (Links.GetElement(LinkSizeInBytes, node) +
300     ↳ Link.LeftAsTargetOffset).GetValue<TLink>();
301
302 protected override TLink GetRightValue(TLink node) =>
303     ↳ (Links.GetElement(LinkSizeInBytes, node) +
304     ↳ Link.RightAsTargetOffset).GetValue<TLink>();
305
306 protected override TLink GetSize(TLink node)
307 {
308     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
309     ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
310     return BitwiseHelpers.PartialRead(previousValue, 5, -5);
311 }
312
313 protected override void SetLeft(TLink node, TLink left) =>
314     ↳ (Links.GetElement(LinkSizeInBytes, node) +
315     ↳ Link.LeftAsTargetOffset).SetValue(left);
316
317 protected override void SetRight(TLink node, TLink right) =>
318     ↳ (Links.GetElement(LinkSizeInBytes, node) +
319     ↳ Link.RightAsTargetOffset).SetValue(right);
320
321 protected override void SetSize(TLink node, TLink size)
322 {
323     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
324     ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
325     (Links.GetElement(LinkSizeInBytes, node) +
326     ↳ Link.SizeAsTargetOffset).SetValue(BitwiseHelpers.PartialWrite(previousValue,
327     ↳ size, 5, -5));
328 }
329
330 protected override bool GetLeftIsChild(TLink node)
331 {
332     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
333     ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
334     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 4, 1);
335 }
336
337 protected override void SetLeftIsChild(TLink node, bool value)
338 {
339     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
340     ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
341     var modified = BitwiseHelpers.PartialWrite(previousValue,
342     ↳ (TLink)(Integer<TLink>)value, 4, 1);
343     (Links.GetElement(LinkSizeInBytes, node) +
344     ↳ Link.SizeAsTargetOffset).SetValue(modified);
345 }
346
347 protected override bool GetRightIsChild(TLink node)
348 {
349     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
350     ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
351     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 3, 1);
352 }
353
354 protected override void SetRightIsChild(TLink node, bool value)
355 {
356     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
357     ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
358     var modified = BitwiseHelpers.PartialWrite(previousValue,
359     ↳ (TLink)(Integer<TLink>)value, 3, 1);
360     (Links.GetElement(LinkSizeInBytes, node) +
361     ↳ Link.SizeAsTargetOffset).SetValue(modified);
362 }

```

```

342     protected override sbyte GetBalance(TLink node)
343     {
344         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
345             ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
346         var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 0,
347             ↪ 3);
348         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
349             ↪ 124 : value & 3);
350         return unpackedValue;
351     }
352
353     protected override void SetBalance(TLink node, sbyte value)
354     {
355         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
356             ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
357         var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
358             ↪ 3);
359         var modified = BitwiseHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
360         (Links.GetElement(LinkSizeInBytes, node) +
361             ↪ Link.SizeAsTargetOffset).SetValue(modified);
362     }
363
364     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
365     {
366         var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
367             ↪ Link.TargetOffset).GetValue<TLink>();
368         var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
369             ↪ Link.TargetOffset).GetValue<TLink>();
370         return LessThan(firstTarget, secondTarget) ||
371             (IsEquals(firstTarget, secondTarget) &&
372             ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
373             ↪ Link.SourceOffset).GetValue<TLink>(),
374             ↪ (Links.GetElement(LinkSizeInBytes, second) +
375             ↪ Link.SourceOffset).GetValue<TLink>()));
376     }
377
378     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
379     {
380         var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
381             ↪ Link.TargetOffset).GetValue<TLink>();
382         var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
383             ↪ Link.TargetOffset).GetValue<TLink>();
384         return GreaterThan(firstTarget, secondTarget) ||
385             (IsEquals(firstTarget, secondTarget) &&
386             ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
387             ↪ Link.SourceOffset).GetValue<TLink>(),
388             ↪ (Links.GetElement(LinkSizeInBytes, second) +
389             ↪ Link.SourceOffset).GetValue<TLink>()));
390     }
391
392     protected override TLink GetTreeRoot() => (Header +
393         ↪ LinksHeader.FirstAsTargetOffset).GetValue<TLink>();
394
395     protected override TLink GetBasePartValue(TLink link) =>
396         ↪ (Links.GetElement(LinkSizeInBytes, link) + Link.TargetOffset).GetValue<TLink>();
397 }
398
399 }

```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs

```

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

```

```

19 {
20     using id = UInt64;
21
22     public unsafe partial class UInt64ResizableDirectMemoryLinks : DisposableBase, ILinks<id>
23     {
24         /// <summary>Возвращает размер одной связи в байтах.</summary>
25         /// <remarks>
26         /// Используется только во вне класса, не рекомендуется использовать внутри.
27         /// Так как во вне не обязательно будет доступен unsafe C#.
28         /// </remarks>
29         public static readonly int LinkSizeInBytes = sizeof(Link);
30
31         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
32
33         private struct Link
34         {
35             public id Source;
36             public id Target;
37             public id LeftAsSource;
38             public id RightAsSource;
39             public id SizeAsSource;
40             public id LeftAsTarget;
41             public id RightAsTarget;
42             public id SizeAsTarget;
43         }
44
45         private struct LinksHeader
46         {
47             public id AllocatedLinks;
48             public id ReservedLinks;
49             public id FreeLinks;
50             public id FirstFreeLink;
51             public id FirstAsSource;
52             public id FirstAsTarget;
53             public id LastFreeLink;
54             public id Reserved8;
55         }
56
57         private readonly long _memoryReservationStep;
58
59         private readonly IResizableDirectMemory _memory;
60         private LinksHeader* _header;
61         private Link* _links;
62
63         private LinksTargetsTreeMethods _targetsTreeMethods;
64         private LinksSourcesTreeMethods _sourcesTreeMethods;
65
66         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
67         // → нужно использовать не список а дерево, так как так можно быстрее проверить на
68         // → наличие связи внутри
69         private UnusedLinksListMethods _unusedLinksListMethods;
70
71         /// <summary>
72         /// Возвращает общее число связей находящихся в хранилище.
73         /// </summary>
74         private id Total => _header->AllocatedLinks - _header->FreeLinks;
75
76         // TODO: Дать возможность переопределять в конструкторе
77         public LinksCombinedConstants<id, id, int> Constants { get; }
78
79         public UInt64ResizableDirectMemoryLinks(string address) : this(address,
80             → DefaultLinksSizeStep) { }
81
82         /// <summary>
83         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
84         /// → минимальным шагом расширения базы данных.
85         /// </summary>
86         /// <param name="address">Полный путь к файлу базы данных.</param>
87         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
88         /// → байтах.</param>
89         public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
90             → this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
91             → memoryReservationStep) { }
92
93         public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
94             → DefaultLinksSizeStep) { }
95
96         public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
97             → memoryReservationStep)
98         {
99             Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
100         }

```



```

91     _memory = memory;
92     _memoryReservationStep = memoryReservationStep;
93     if (memory.ReservedCapacity < memoryReservationStep)
94     {
95         memory.ReservedCapacity = memoryReservationStep;
96     }
97     SetPointers(_memory);
98     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
99     _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) +
100     ↪ sizeof(LinksHeader);
101     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
102     _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(LinksHeader)) /
103     ↪ sizeof(Link));
104 }
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 public id Count(IList<id> restrictions)
108 {
109     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
110     if (restrictions.Count == 0)
111     {
112         return Total;
113     }
114     if (restrictions.Count == 1)
115     {
116         var index = restrictions[Constants.IndexPart];
117         if (index == Constants.Any)
118         {
119             return Total;
120         }
121         return Exists(index) ? 1UL : 0UL;
122     }
123     if (restrictions.Count == 2)
124     {
125         var index = restrictions[Constants.IndexPart];
126         var value = restrictions[1];
127         if (index == Constants.Any)
128         {
129             if (value == Constants.Any)
130             {
131                 return Total; // Any - как отсутствие ограничения
132             }
133             return _sourcesTreeMethods.CalculateReferences(value)
134                 + _targetsTreeMethods.CalculateReferences(value);
135         }
136         else
137         {
138             if (!Exists(index))
139             {
140                 return 0;
141             }
142             if (value == Constants.Any)
143             {
144                 return 1;
145             }
146             var storedLinkValue = GetLinkUnsafe(index);
147             if (storedLinkValue->Source == value ||
148                 storedLinkValue->Target == value)
149             {
150                 return 1;
151             }
152             return 0;
153         }
154     }
155     if (restrictions.Count == 3)
156     {
157         var index = restrictions[Constants.IndexPart];
158         var source = restrictions[Constants.SourcePart];
159         var target = restrictions[Constants.TargetPart];
160         if (index == Constants.Any)
161         {
162             if (source == Constants.Any && target == Constants.Any)
163             {
164                 return Total;
165             }
166             else if (source == Constants.Any)
167             {
168                 return _targetsTreeMethods.CalculateReferences(target);
169             }
170             else if (target == Constants.Any)
171             {
172                 return _sourcesTreeMethods.CalculateReferences(source);
173             }
174             else
175             {
176                 return Exists(index, source, target) ? 1UL : 0UL;
177             }
178         }
179         else
180         {
181             if (!Exists(index))
182             {
183                 return 0;
184             }
185             var linkValue = GetLinkUnsafe(index);
186             if (linkValue->Source == source && linkValue->Target == target)
187             {
188                 return 1;
189             }
190             return 0;
191         }
192     }
193     return 0;
194 }

```

```

167     }
168     else if (target == Constants.Any)
169     {
170         return _sourcesTreeMethods.CalculateReferences(source);
171     }
172     else //if(source != Any && target != Any)
173     {
174         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
175         var link = _sourcesTreeMethods.Search(source, target);
176         return link == Constants.Null ? OUL : 1UL;
177     }
178 }
179 else
180 {
181     if (!Exists(index))
182     {
183         return 0;
184     }
185     if (source == Constants.Any && target == Constants.Any)
186     {
187         return 1;
188     }
189     var storedLinkValue = GetLinkUnsafe(index);
190     if (source != Constants.Any && target != Constants.Any)
191     {
192         if (storedLinkValue->Source == source &&
193             storedLinkValue->Target == target)
194         {
195             return 1;
196         }
197         return 0;
198     }
199     var value = default(id);
200     if (source == Constants.Any)
201     {
202         value = target;
203     }
204     if (target == Constants.Any)
205     {
206         value = source;
207     }
208     if (storedLinkValue->Source == value ||
209         storedLinkValue->Target == value)
210     {
211         return 1;
212     }
213     return 0;
214 }
215 }
216 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
217 }
218
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
221 {
222     if (restrictions.Count == 0)
223     {
224         for (id link = 1; link <= _header->AllocatedLinks; link++)
225         {
226             if (Exists(link))
227             {
228                 if (handler(GetLinkStruct(link)) == Constants.Break)
229                 {
230                     return Constants.Break;
231                 }
232             }
233         }
234         return Constants.Continue;
235     }
236     if (restrictions.Count == 1)
237     {
238         var index = restrictions[Constants.IndexPart];
239         if (index == Constants.Any)
240         {
241             return Each(handler, ArrayPool<ulong>.Empty);
242         }
243         if (!Exists(index))
244         {

```

```

245         return Constants.Continue;
246     }
247     return handler(GetLinkStruct(index));
248 }
249 if (restrictions.Count == 2)
250 {
251     var index = restrictions[Constants.IndexPart];
252     var value = restrictions[1];
253     if (index == Constants.Any)
254     {
255         if (value == Constants.Any)
256         {
257             return Each(handler, ArrayPool<ulong>.Empty);
258         }
259         if (Each(handler, new[] { index, value, Constants.Any }) == Constants.Break)
260         {
261             return Constants.Break;
262         }
263         return Each(handler, new[] { index, Constants.Any, value });
264     }
265     else
266     {
267         if (!Exists(index))
268         {
269             return Constants.Continue;
270         }
271         if (value == Constants.Any)
272         {
273             return handler(GetLinkStruct(index));
274         }
275         var storedLinkValue = GetLinkUnsafe(index);
276         if (storedLinkValue->Source == value ||
277             storedLinkValue->Target == value)
278         {
279             return handler(GetLinkStruct(index));
280         }
281         return Constants.Continue;
282     }
283 }
284 if (restrictions.Count == 3)
285 {
286     var index = restrictions[Constants.IndexPart];
287     var source = restrictions[Constants.SourcePart];
288     var target = restrictions[Constants.TargetPart];
289     if (index == Constants.Any)
290     {
291         if (source == Constants.Any && target == Constants.Any)
292         {
293             return Each(handler, ArrayPool<ulong>.Empty);
294         }
295         else if (source == Constants.Any)
296         {
297             return _targetsTreeMethods.EachReference(target, handler);
298         }
299         else if (target == Constants.Any)
300         {
301             return _sourcesTreeMethods.EachReference(source, handler);
302         }
303         else //if(source != Any && target != Any)
304         {
305             var link = _sourcesTreeMethods.Search(source, target);
306             return link == Constants.Null ? Constants.Continue :
307                 ↪ handler(GetLinkStruct(link));
308         }
309     }
310     else
311     {
312         if (!Exists(index))
313         {
314             return Constants.Continue;
315         }
316         if (source == Constants.Any && target == Constants.Any)
317         {
318             return handler(GetLinkStruct(index));
319         }
320         var storedLinkValue = GetLinkUnsafe(index);
321         if (source != Constants.Any && target != Constants.Any)
322         {

```

```

322         if (storedLinkValue->Source == source &&
323             storedLinkValue->Target == target)
324         {
325             return handler(GetLinkStruct(index));
326         }
327         return Constants.Continue;
328     }
329     var value = default(id);
330     if (source == Constants.Any)
331     {
332         value = target;
333     }
334     if (target == Constants.Any)
335     {
336         value = source;
337     }
338     if (storedLinkValue->Source == value ||
339         storedLinkValue->Target == value)
340     {
341         return handler(GetLinkStruct(index));
342     }
343     return Constants.Continue;
344 }
345 }
346 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
347 }
348
349 /// <remarks>
350 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
351 /// </remarks>
352 [MethodImpl(MethodImplOptions.AggressiveInlining)]
353 public id Update(IList<id> values)
354 {
355     var linkIndex = values[Constants.IndexPart];
356     var link = GetLinkUnsafe(linkIndex);
357     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
358     if (link->Source != Constants.Null)
359     {
360         _sourcesTreeMethods.Detach(new IntPtr(&_header->FirstAsSource), linkIndex);
361     }
362     if (link->Target != Constants.Null)
363     {
364         _targetsTreeMethods.Detach(new IntPtr(&_header->FirstAsTarget), linkIndex);
365     }
366     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
367     var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
368     var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
369     if (leftTreeSize != rightTreeSize)
370     {
371         throw new Exception("One of the trees is broken.");
372     }
373     #endif
374     link->Source = values[Constants.SourcePart];
375     link->Target = values[Constants.TargetPart];
376     if (link->Source != Constants.Null)
377     {
378         _sourcesTreeMethods.Attach(new IntPtr(&_header->FirstAsSource), linkIndex);
379     }
380     if (link->Target != Constants.Null)
381     {
382         _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
383     }
384     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
385     leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
386     rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
387     if (leftTreeSize != rightTreeSize)
388     {
389         throw new Exception("One of the trees is broken.");
390     }
391     #endif
392     return linkIndex;
393 }
394
395 [MethodImpl(MethodImplOptions.AggressiveInlining)]
396 private IList<id> GetLinkStruct(id linkIndex)

```

```

397 {
398     var link = GetLinkUnsafe(linkIndex);
399     return new UInt64Link(linkIndex, link->Source, link->Target);
400 }
401
402 [MethodImpl(MethodImplOptions.AggressiveInlining)]
403 private Link* GetLinkUnsafe(id linkIndex) => &_links[linkIndex];
404
405 /// <remarks>
406 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
407   ↳ пространство
408 /// </remarks>
409 public id Create()
410 {
411     var freeLink = _header->FirstFreeLink;
412     if (freeLink != Constants.Null)
413     {
414         _unusedLinksListMethods.Detach(freeLink);
415     }
416     else
417     {
418         if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
419         {
420             throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
421         }
422         if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
423         {
424             _memory.ReservedCapacity += _memory.ReservationStep;
425             SetPointers(_memory);
426             _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
427         }
428         _header->AllocatedLinks++;
429         _memory.UsedCapacity += sizeof(Link);
430         freeLink = _header->AllocatedLinks;
431     }
432     return freeLink;
433 }
434
435 public void Delete(id link)
436 {
437     if (link < _header->AllocatedLinks)
438     {
439         _unusedLinksListMethods.AttachAsFirst(link);
440     }
441     else if (link == _header->AllocatedLinks)
442     {
443         _header->AllocatedLinks--;
444         _memory.UsedCapacity -= sizeof(Link);
445         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
446         //   ↳ пока не дойдём до первой существующей связи
447         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
448         while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
449         {
450             _unusedLinksListMethods.Detach(_header->AllocatedLinks);
451             _header->AllocatedLinks--;
452             _memory.UsedCapacity -= sizeof(Link);
453         }
454     }
455 }
456
457 /// <remarks>
458 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
459   ↳ адрес реально поменялся
460 ///
461 /// Указатель this.links может быть в том же месте,
462 /// так как 0-я связь не используется и имеет такой же размер как Header,
463 /// поэтому header размещается в том же месте, что и 0-я связь
464 /// </remarks>
465 private void SetPointers(IResizableDirectMemory memory)
466 {
467     if (memory == null)
468     {
469         _header = null;
470         _links = null;
471         _unusedLinksListMethods = null;
472         _targetsTreeMethods = null;
473         _unusedLinksListMethods = null;
474     }
475     else

```

```

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

```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs

```

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

```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using System.Text;
5 using Platform.Collections.Methods.Trees;
6 using Platform.Data.Constants;
7
8 namespace Platform.Data.Doublets.ResizableDirectMemory
9 {
10     unsafe partial class UInt64ResizableDirectMemoryLinks
11     {
12         private abstract class LinksTreeMethodsBase :
13             ↳ SizedAndThreadedAVLBalancedTreeMethods<ulong>
14         {
15             private readonly UInt64ResizableDirectMemoryLinks _memory;
16             private readonly LinksCombinedConstants<ulong, ulong, int> _constants;
17             protected readonly Link* Links;
18             protected readonly LinksHeader* Header;
19
20             protected LinksTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory)
21             {
22                 Links = memory._links;
23                 Header = memory._header;
24                 _memory = memory;
25                 _constants = memory.Constants;
26             }
27
28             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29             protected abstract ulong GetTreeRoot();
30
31             [MethodImpl(MethodImplOptions.AggressiveInlining)]
32             protected abstract ulong GetBasePartValue(ulong link);
33
34             public ulong this[ulong index]
35             {
36                 get
37                 {
38                     var root = GetTreeRoot();
39                     if (index >= GetSize(root))
40                     {
41                         return 0;
42                     }
43                     while (root != 0)
44                     {
45                         var left = GetLeftOrDefault(root);
46                         var leftSize = GetSizeOrZero(left);
47                         if (index < leftSize)
48                         {
49                             root = left;
50                             continue;
51                         }
52                         if (index == leftSize)
53                         {
54                             return root;
55                         }
56                         root = GetRightOrDefault(root);
57                         index -= leftSize + 1;
58                     }
59                     return 0; // TODO: Impossible situation exception (only if tree structure
60                             ↳ broken)
61                 }
62             }
63
64             // TODO: Return indices range instead of references count
65             public ulong CalculateReferences(ulong link)
66             {
67                 var root = GetTreeRoot();
68                 var total = GetSize(root);
69                 var totalRightIgnore = OUL;
70                 while (root != 0)
71                 {
72                     var @base = GetBasePartValue(root);
73                     if (@base <= link)
74                     {
75                         root = GetRightOrDefault(root);
76                     }
77                     else
78                     {
79                         totalRightIgnore += GetRightSize(root) + 1;
80                     }
81                 }
82                 return totalRightIgnore;
83             }
84         }
85     }
86 }
```

```

78         root = GetLeftOrDefault(root);
79     }
80 }
81 root = GetTreeRoot();
82 var totalLeftIgnore = 0UL;
83 while (root != 0)
84 {
85     var @base = GetBasePartValue(root);
86     if (@base >= link)
87     {
88         root = GetLeftOrDefault(root);
89     }
90     else
91     {
92         totalLeftIgnore += GetLeftSize(root) + 1;
93         root = GetRightOrDefault(root);
94     }
95 }
96 return total - totalRightIgnore - totalLeftIgnore;
97 }
98
99 public ulong EachReference(ulong link, Func<IList<ulong>, ulong> handler)
100 {
101     var root = GetTreeRoot();
102     if (root == 0)
103     {
104         return _constants.Continue;
105     }
106     ulong first = 0, current = root;
107     while (current != 0)
108     {
109         var @base = GetBasePartValue(current);
110         if (@base >= link)
111         {
112             if (@base == link)
113             {
114                 first = current;
115             }
116             current = GetLeftOrDefault(current);
117         }
118         else
119         {
120             current = GetRightOrDefault(current);
121         }
122     }
123     if (first != 0)
124     {
125         current = first;
126         while (true)
127         {
128             if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
129             {
130                 return _constants.Break;
131             }
132             current = GetNext(current);
133             if (current == 0 || GetBasePartValue(current) != link)
134             {
135                 break;
136             }
137         }
138     }
139     return _constants.Continue;
140 }
141
142 protected override void PrintNodeValue(ulong node, StringBuilder sb)
143 {
144     sb.Append(' ');
145     sb.Append(Links[node].Source);
146     sb.Append('-');
147     sb.Append('>');
148     sb.Append(Links[node].Target);
149 }
150
151 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
152 {
153     public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)
154         : base(memory)
155     {
156

```



```

157     }
158
159     protected override IntPtr GetLeftPointer(ulong node) => new
160     ↪ IntPtr(&Links[node].LeftAsSource);
161
162     protected override IntPtr GetRightPointer(ulong node) => new
163     ↪ IntPtr(&Links[node].RightAsSource);
164
165     protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsSource;
166
167     protected override ulong GetRightValue(ulong node) => Links[node].RightAsSource;
168
169     protected override ulong GetSize(ulong node)
170     {
171         var previousValue = Links[node].SizeAsSource;
172         //return MathHelpers.PartialRead(previousValue, 5, -5);
173         return (previousValue & 4294967264) >> 5;
174     }
175
176     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource
177     ↪ = left;
178
179     protected override void SetRight(ulong node, ulong right) =>
180     ↪ Links[node].RightAsSource = right;
181
182     protected override void SetSize(ulong node, ulong size)
183     {
184         var previousValue = Links[node].SizeAsSource;
185         //var modified = MathHelpers.PartialWrite(previousValue, size, 5, -5);
186         var modified = (previousValue & 31) | ((size & 134217727) << 5);
187         Links[node].SizeAsSource = modified;
188     }
189
190     protected override bool GetLeftIsChild(ulong node)
191     {
192         var previousValue = Links[node].SizeAsSource;
193         //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
194         return (previousValue & 16) >> 4 == 1UL;
195     }
196
197     protected override void SetLeftIsChild(ulong node, bool value)
198     {
199         var previousValue = Links[node].SizeAsSource;
200         //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value,
201         ↪ 4, 1);
202         var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
203         Links[node].SizeAsSource = modified;
204     }
205
206     protected override bool GetRightIsChild(ulong node)
207     {
208         var previousValue = Links[node].SizeAsSource;
209         //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
210         return (previousValue & 8) >> 3 == 1UL;
211     }
212
213     protected override void SetRightIsChild(ulong node, bool value)
214     {
215         var previousValue = Links[node].SizeAsSource;
216         //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value,
217         ↪ 3, 1);
218         var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
219         Links[node].SizeAsSource = modified;
220     }
221
222     protected override sbyte GetBalance(ulong node)
223     {
224         var previousValue = Links[node].SizeAsSource;
225         //var value = MathHelpers.PartialRead(previousValue, 0, 3);
226         var value = previousValue & 7;
227         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
228         ↪ 124 : value & 3);
229         return unpackedValue;
230     }
231
232     protected override void SetBalance(ulong node, sbyte value)
233     {
234         var previousValue = Links[node].SizeAsSource;

```

```

228     var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
229     //var modified = MathHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
230     var modified = (previousValue & 4294967288) | (packagedValue & 7);
231     Links[node].SizeAsSource = modified;
232 }
233
234 protected override bool FirstIsToLeftOfSecond(ulong first, ulong second)
235     => Links[first].Source < Links[second].Source ||
236     (Links[first].Source == Links[second].Source && Links[first].Target <
237     ↪ Links[second].Target);
238
239 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
240     => Links[first].Source > Links[second].Source ||
241     (Links[first].Source == Links[second].Source && Links[first].Target >
242     ↪ Links[second].Target);
243
244 protected override ulong GetTreeRoot() => Header->FirstAsSource;
245
246 protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
247
248 /// <summary>
249 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
250 ↪ (концом)
251 /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
252 /// </summary>
253 /// <param name="source">Индекс связи, которая является началом на искомой
254 ↪ связи.</param>
255 /// <param name="target">Индекс связи, которая является концом на искомой
256 ↪ связи.</param>
257 /// <returns>Индекс искомой связи.</returns>
258 public ulong Search(ulong source, ulong target)
259 {
260     var root = Header->FirstAsSource;
261     while (root != 0)
262     {
263         var rootSource = Links[root].Source;
264         var rootTarget = Links[root].Target;
265         if (FirstIsToLeftOfSecond(source, target, rootSource, rootTarget)) //
266             ↪ node.Key < root.Key
267         {
268             root = GetLeftOrDefault(root);
269         }
270         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
271             ↪ // node.Key > root.Key
272         {
273             root = GetRightOrDefault(root);
274         }
275         else // node.Key == root.Key
276         {
277             return root;
278         }
279     }
280     return 0;
281 }
282
283 [MethodImpl(MethodImplOptions.AggressiveInlining)]
284 private static bool FirstIsToLeftOfSecond(ulong firstSource, ulong firstTarget,
285     ↪ ulong secondSource, ulong secondTarget)
286     => firstSource < secondSource || (firstSource == secondSource && firstTarget <
287     ↪ secondTarget);
288
289 [MethodImpl(MethodImplOptions.AggressiveInlining)]
290 private static bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
291     ↪ ulong secondSource, ulong secondTarget)
292     => firstSource > secondSource || (firstSource == secondSource && firstTarget >
293     ↪ secondTarget);
294
295 [MethodImpl(MethodImplOptions.AggressiveInlining)]
296 protected override void ClearNode(ulong node)
297 {
298     Links[node].LeftAsSource = OUL;
299     Links[node].RightAsSource = OUL;
300     Links[node].SizeAsSource = OUL;
301 }
302
303 [MethodImpl(MethodImplOptions.AggressiveInlining)]
304 protected override ulong GetZero() => OUL;
305

```

```

295 [MethodImpl(MethodImplOptions.AggressiveInlining)]
296 protected override ulong GetOne() => 1UL;
297
298 [MethodImpl(MethodImplOptions.AggressiveInlining)]
299 protected override ulong GetTwo() => 2UL;
300
301 [MethodImpl(MethodImplOptions.AggressiveInlining)]
302 protected override bool ValueEqualToZero(IntPtr pointer) =>
303     ↳ *(ulong*)pointer.ToPointer() == 0UL;
304
305 [MethodImpl(MethodImplOptions.AggressiveInlining)]
306 protected override bool EqualToZero(ulong value) => value == 0UL;
307
308 [MethodImpl(MethodImplOptions.AggressiveInlining)]
309 protected override bool IsEquals(ulong first, ulong second) => first == second;
310
311 [MethodImpl(MethodImplOptions.AggressiveInlining)]
312 protected override bool GreaterThanZero(ulong value) => value > 0UL;
313
314 [MethodImpl(MethodImplOptions.AggressiveInlining)]
315 protected override bool GreaterThan(ulong first, ulong second) => first > second;
316
317 [MethodImpl(MethodImplOptions.AggressiveInlining)]
318 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >=
319     ↳ second;
320
321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
322 protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0
323     ↳ is always true for ulong
324
325 [MethodImpl(MethodImplOptions.AggressiveInlining)]
326 protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is
327     ↳ always >= 0 for ulong
328
329 [MethodImpl(MethodImplOptions.AggressiveInlining)]
330 protected override bool LessOrEqualThan(ulong first, ulong second) => first <=
331     ↳ second;
332
333 [MethodImpl(MethodImplOptions.AggressiveInlining)]
334 protected override bool LessThanZero(ulong value) => false; // value < 0 is always
335     ↳ false for ulong
336
337 [MethodImpl(MethodImplOptions.AggressiveInlining)]
338 protected override bool LessThan(ulong first, ulong second) => first < second;
339
340 [MethodImpl(MethodImplOptions.AggressiveInlining)]
341 protected override ulong Increment(ulong value) => ++value;
342
343 [MethodImpl(MethodImplOptions.AggressiveInlining)]
344 protected override ulong Decrement(ulong value) => --value;
345
346 [MethodImpl(MethodImplOptions.AggressiveInlining)]
347 protected override ulong Add(ulong first, ulong second) => first + second;
348
349 [MethodImpl(MethodImplOptions.AggressiveInlining)]
350 protected override ulong Subtract(ulong first, ulong second) => first - second;
351 }
352
353 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
354 {
355     public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
356         : base(memory)
357     {
358     }
359
360     //protected override IntPtr GetLeft(ulong node) => new
361     ↳ IntPtr(&Links[node].LeftAsTarget);
362
363     //protected override IntPtr GetRight(ulong node) => new
364     ↳ IntPtr(&Links[node].RightAsTarget);
365
366     //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
367
368     //protected override void SetLeft(ulong node, ulong left) =>
369     ↳ Links[node].LeftAsTarget = left;
370
371     //protected override void SetRight(ulong node, ulong right) =>
372     ↳ Links[node].RightAsTarget = right;

```

```

364 //protected override void SetSize(ulong node, ulong size) =>
365     ↳ Links[node].SizeAsTarget = size;
366
367 protected override IntPtr GetLeftPointer(ulong node) => new
368     ↳ IntPtr(&Links[node].LeftAsTarget);
369
370 protected override IntPtr GetRightPointer(ulong node) => new
371     ↳ IntPtr(&Links[node].RightAsTarget);
372
373 protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsTarget;
374
375 protected override ulong GetRightValue(ulong node) => Links[node].RightAsTarget;
376
377 protected override ulong GetSize(ulong node)
378 {
379     var previousValue = Links[node].SizeAsTarget;
380     //return MathHelpers.PartialRead(previousValue, 5, -5);
381     return (previousValue & 4294967264) >> 5;
382 }
383
384 protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget
385     ↳ = left;
386
387 protected override void SetRight(ulong node, ulong right) =>
388     ↳ Links[node].RightAsTarget = right;
389
390 protected override void SetSize(ulong node, ulong size)
391 {
392     var previousValue = Links[node].SizeAsTarget;
393     //var modified = MathHelpers.PartialWrite(previousValue, size, 5, -5);
394     var modified = (previousValue & 31) | ((size & 134217727) << 5);
395     Links[node].SizeAsTarget = modified;
396 }
397
398 protected override bool GetLeftIsChild(ulong node)
399 {
400     var previousValue = Links[node].SizeAsTarget;
401     //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
402     return (previousValue & 16) >> 4 == 1UL;
403     // TODO: Check if this is possible to use
404     //var nodeSize = GetSize(node);
405     //var left = GetLeftValue(node);
406     //var leftSize = GetSizeOrZero(left);
407     //return leftSize > 0 && nodeSize > leftSize;
408 }
409
410 protected override void SetLeftIsChild(ulong node, bool value)
411 {
412     var previousValue = Links[node].SizeAsTarget;
413     //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value,
414     ↳ 4, 1);
415     var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
416     Links[node].SizeAsTarget = modified;
417 }
418
419 protected override bool GetRightIsChild(ulong node)
420 {
421     var previousValue = Links[node].SizeAsTarget;
422     //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
423     return (previousValue & 8) >> 3 == 1UL;
424     // TODO: Check if this is possible to use
425     //var nodeSize = GetSize(node);
426     //var right = GetRightValue(node);
427     //var rightSize = GetSizeOrZero(right);
428     //return rightSize > 0 && nodeSize > rightSize;
429 }
430
431 protected override void SetRightIsChild(ulong node, bool value)
432 {
433     var previousValue = Links[node].SizeAsTarget;
434     //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value,
435     ↳ 3, 1);
436     var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
437     Links[node].SizeAsTarget = modified;
438 }
439
440 protected override sbyte GetBalance(ulong node)
441 {

```

```

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

```

./Sequences/Converters/BalancedVariantConverter.cs

```

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

```

```

36     }
37
38     private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
39     {
40         var loopedLength = length - (length % 2);
41         for (var i = 0; i < loopedLength; i += 2)
42         {
43             destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
44         }
45         if (length > loopedLength)
46         {
47             destination[length / 2] = source[length - 1];
48         }
49     }
50 }
51 }

```

./Sequences/Converters/CompressingConverter.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Collections;
6  using Platform.Helpers.Singletons;
7  using Platform.Numbers;
8  using Platform.Data.Constants;
9  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
10
11 namespace Platform.Data.Doublets.Sequences.Converters
12 {
13     /// <remarks>
14     /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
15     /// ↳ Links на этапе сжатия.
16     /// А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
17     /// ↳ таком случае тип значения элемента массива может быть любым, как char так и ulong.
18     /// Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
19     /// ↳ пар, а так же разом выполнить замену.
20     /// </remarks>
21     public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
22     {
23         private static readonly LinksCombinedConstants<bool, TLink, long> _constants =
24             ↳ Default<LinksCombinedConstants<bool, TLink, long>>.Instance;
25         private static readonly EqualityComparer<TLink> _equalityComparer =
26             ↳ EqualityComparer<TLink>.Default;
27         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
28
29         private readonly IConverter<IList<TLink>, TLink> _baseConverter;
30         private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
31         private readonly TLink _minFrequencyToCompress;
32         private readonly bool _doInitialFrequenciesIncrement;
33         private Doublet<TLink> _maxDoublet;
34         private LinkFrequency<TLink> _maxDoubletData;
35
36         private struct HalfDoublet
37         {
38             public TLink Element;
39             public LinkFrequency<TLink> DoubletData;
40
41             public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
42             {
43                 Element = element;
44                 DoubletData = doubletData;
45             }
46
47             public override string ToString() => $"{Element}: ({DoubletData})";
48         }
49
50         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
51             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
52             : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
53         {
54         }
55
56         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
57             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
58             ↳ doInitialFrequenciesIncrement)
59             : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
60                 ↳ doInitialFrequenciesIncrement)
61         {
62         }
63     }
64 }

```

```

54 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
55     ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
56     ↳ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
57     : base(links)
58 {
59     _baseConverter = baseConverter;
60     _doubletFrequenciesCache = doubletFrequenciesCache;
61     if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
62     {
63         minFrequencyToCompress = Integer<TLink>.One;
64     }
65     _minFrequencyToCompress = minFrequencyToCompress;
66     _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
67     ResetMaxDoublet();
68 }
69
70 public override TLink Convert(IList<TLink> source) =>
71     ↳ _baseConverter.Convert(Compress(source));
72
73 /// <remarks>
74 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
75 /// Faster version (doublets' frequencies dictionary is not recreated).
76 /// </remarks>
77 private IList<TLink> Compress(IList<TLink> sequence)
78 {
79     if (sequence.IsNullOrEmpty())
80     {
81         return null;
82     }
83     if (sequence.Count == 1)
84     {
85         return sequence;
86     }
87     if (sequence.Count == 2)
88     {
89         return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
90     }
91     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
92     var copy = new HalfDoublet[sequence.Count];
93     Doublet<TLink> doublet = default;
94     for (var i = 1; i < sequence.Count; i++)
95     {
96         doublet.Source = sequence[i - 1];
97         doublet.Target = sequence[i];
98         LinkFrequency<TLink> data;
99         if (_doInitialFrequenciesIncrement)
100         {
101             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
102         }
103         else
104         {
105             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
106             if (data == null)
107             {
108                 throw new NotSupportedException("If you ask not to increment
109                     ↳ frequencies, it is expected that all frequencies for the sequence
110                     ↳ are prepared.");
111             }
112         }
113         copy[i - 1].Element = sequence[i - 1];
114         copy[i - 1].DoubletData = data;
115         UpdateMaxDoublet(ref doublet, data);
116     }
117     copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
118     copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
119     if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
120     {
121         var newLength = ReplaceDoublets(copy);
122         sequence = new TLink[newLength];
123         for (int i = 0; i < newLength; i++)
124         {
125             sequence[i] = copy[i].Element;
126         }
127     }
128     return sequence;
129 }
130
131 /// <remarks>

```

```

128 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
129 /// </remarks>
130 private int ReplaceDoublets(HalfDoublet[] copy)
131 {
132     var oldLength = copy.Length;
133     var newLength = copy.Length;
134     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
135     {
136         var maxDoubletSource = _maxDoublet.Source;
137         var maxDoubletTarget = _maxDoublet.Target;
138         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
139         {
140             _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
141         }
142         var maxDoubletReplacementLink = _maxDoubletData.Link;
143         oldLength--;
144         var oldLengthMinusTwo = oldLength - 1;
145         // Substitute all usages
146         int w = 0, r = 0; // (r == read, w == write)
147         for (; r < oldLength; r++)
148         {
149             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
150                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
151             {
152                 if (r > 0)
153                 {
154                     var previous = copy[w - 1].Element;
155                     copy[w - 1].DoubletData.DecrementFrequency();
156                     copy[w - 1].DoubletData =
157                         ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
158                             ↪ maxDoubletReplacementLink);
159                 }
160                 if (r < oldLengthMinusTwo)
161                 {
162                     var next = copy[r + 2].Element;
163                     copy[r + 1].DoubletData.DecrementFrequency();
164                     copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
165                         ↪ next);
166                 }
167                 copy[w++].Element = maxDoubletReplacementLink;
168                 r++;
169                 newLength--;
170             }
171             else
172             {
173                 copy[w++] = copy[r];
174             }
175         }
176         oldLength = newLength;
177         ResetMaxDoublet();
178         UpdateMaxDoublet(copy, newLength);
179     }
180     return newLength;
181 }
182
183 [MethodImpl(MethodImplOptions.AggressiveInlining)]
184 private void ResetMaxDoublet()
185 {
186     _maxDoublet = new Doublet<TLink>();
187     _maxDoubletData = new LinkFrequency<TLink>();
188 }
189
190 [MethodImpl(MethodImplOptions.AggressiveInlining)]
191 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
192 {
193     Doublet<TLink> doublet = default;
194     for (var i = 1; i < length; i++)
195     {
196         doublet.Source = copy[i - 1].Element;
197         doublet.Target = copy[i].Element;
198         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
199     }
200 }
201

```



```

202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
204 {
205     var frequency = data.Frequency;
206     var maxFrequency = _maxDoubletData.Frequency;
207     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
208     ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
209     ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
210     ↪ _maxDoublet.Target)))
211     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
212     ↪ (_comparer.Compare(maxFrequency, frequency) < 0 ||
213     ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
214     ↪ _comparer.Compare(ArithmeticHelpers.Add(doublet.Source, doublet.Target),
215     ↪ ArithmeticHelpers.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /*
216     ↪ gives better stability and better compression on sequent data and even on
217     ↪ rundom numbers data (but gives collisions anyway) */
218     {
219         _maxDoublet = doublet;
220         _maxDoubletData = data;
221     }
222 }
223 }
224 }
225 }
226 }

```

./Sequences/Converters/LinksListToSequenceConverterBase.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Sequences.Converters
5 {
6     public abstract class LinksListToSequenceConverterBase<TLink> : IConverter<IList<TLink>,
7     ↪ TLink>
8     {
9         protected readonly ILinks<TLink> Links;
10        public LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
11        public abstract TLink Convert(IList<TLink> source);
12    }
13 }

```

./Sequences/Converters/OptimalVariantConverter.cs

```

1 using System.Collections.Generic;
2 using System.Linq;
3 using Platform.Interfaces;
4
5 namespace Platform.Data.Doublets.Sequences.Converters
6 {
7     public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
8     {
9         private static readonly EqualityComparer<TLink> _equalityComparer =
10        ↪ EqualityComparer<TLink>.Default;
11        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13        private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
14
15        public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
16        ↪ sequenceToItsLocalElementLevelsConverter) : base(links)
17        => _sequenceToItsLocalElementLevelsConverter =
18        ↪ sequenceToItsLocalElementLevelsConverter;
19
20        public override TLink Convert(IList<TLink> sequence)
21        {
22            var length = sequence.Count;
23            if (length == 1)
24            {
25                return sequence[0];
26            }
27            var links = Links;
28            if (length == 2)
29            {
30                return links.GetOrCreate(sequence[0], sequence[1]);
31            }
32            sequence = sequence.ToArray();
33            var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
34            while (length > 2)
35            {
36                var levelRepeat = 1;
37                var currentLevel = levels[0];
38                var previousLevel = levels[0];
39                var skipOnce = false;

```

```

37     var w = 0;
38     for (var i = 1; i < length; i++)
39     {
40         if (_equalityComparer.Equals(currentLevel, levels[i]))
41         {
42             levelRepeat++;
43             skipOnce = false;
44             if (levelRepeat == 2)
45             {
46                 sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
47                 var newLevel = i >= length - 1 ?
48                     GetPreviousLowerThanCurrentOrCurrent(previousLevel,
49                     ↪ currentLevel) :
50                     i < 2 ?
51                     GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
52                     GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
53                     ↪ currentLevel, levels[i + 1]);
54                 levels[w] = newLevel;
55                 previousLevel = currentLevel;
56                 w++;
57                 levelRepeat = 0;
58                 skipOnce = true;
59             }
60             else if (i == length - 1)
61             {
62                 sequence[w] = sequence[i];
63                 levels[w] = levels[i];
64                 w++;
65             }
66         }
67         else
68         {
69             currentLevel = levels[i];
70             levelRepeat = 1;
71             if (skipOnce)
72             {
73                 skipOnce = false;
74             }
75             else
76             {
77                 sequence[w] = sequence[i - 1];
78                 levels[w] = levels[i - 1];
79                 previousLevel = levels[w];
80                 w++;
81             }
82             if (i == length - 1)
83             {
84                 sequence[w] = sequence[i];
85                 levels[w] = levels[i];
86                 w++;
87             }
88         }
89     }
90     length = w;
91     return links.GetOrCreate(sequence[0], sequence[1]);
92 }
93 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
94 ↪ current, TLink next)
95 {
96     return _comparer.Compare(previous, next) > 0
97         ? _comparer.Compare(previous, current) < 0 ? previous : current
98         : _comparer.Compare(next, current) < 0 ? next : current;
99 }
100 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
101     ↪ _comparer.Compare(next, current) < 0 ? next : current;
102 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
103     ↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
104 }

```

./Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs

```

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

```

```

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

```

./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs

```

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

```

./Sequences/CreteriaMatchers/MarkedSequenceCreteriaMatcher.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
5 {
6     public class MarkedSequenceCreteriaMatcher<TLink> : ICreteriaMatcher<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(sequenceCandidate), _sequenceMarkerLink)
21        || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
        ↳ 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), default))
25             {
26                 var source = Links.GetSource(cursor);
27                 var target = Links.GetTarget(cursor);
28                 if (_equalityComparer.Equals(_heightProvider.Get(source),
29                     ↳ _heightProvider.Get(target)))
30                 {
31                     break;
32                 }
33                 else
34                 {
35                     _stack.Push(source);
36                     cursor = target;
37                 }
38             }
39             var left = cursor;
40             var right = appendant;
41             while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
42             {
43                 right = Links.GetOrCreate(left, right);
44                 left = cursor;
45             }
46             return Links.GetOrCreate(left, right);
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<IList<TLink>,
            ↳ IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
            ↳ duplicateFragmentsProvider;
11         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
12     }
13 }

```

./Sequences/DuplicateSegmentsProvider.cs

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using Platform.Interfaces;
5  using Platform.Collections;
6  using Platform.Collections.Lists;
7  using Platform.Collections.Segments;
8  using Platform.Collections.Segments.Walkers;
9  using Platform.Helpers;
10 using Platform.Helpers.Singletons;
11 using Platform.Numbers;
12 using Platform.Data.Sequences;
13
14 namespace Platform.Data.Doublets.Sequences
15 {

```

```

16 public class DuplicateSegmentsProvider<TLink> :
    ↳ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
    ↳ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
17 {
18     private readonly IList<TLink> _links;
19     private readonly ISequences<TLink> _sequences;
20     private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
21     private BitString _visited;
22
23     private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
    ↳ IList<TLink>>>
24     {
25         private readonly IListEqualityComparer<TLink> _listComparer;
26         public ItemEquilityComparer() => _listComparer =
    ↳ Default<IListEqualityComparer<TLink>>.Instance;
27         public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
    ↳ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
    ↳ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
    ↳ right.Value);
28         public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
    ↳ HashHelpers.Generate(_listComparer.GetHashCode(pair.Key),
    ↳ _listComparer.GetHashCode(pair.Value));
29     }
30
31     private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
32     {
33         private readonly IListComparer<TLink> _listComparer;
34
35         public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
36
37         public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
    ↳ KeyValuePair<IList<TLink>, IList<TLink>> right)
38         {
39             var intermediateResult = _listComparer.Compare(left.Key, right.Key);
40             if (intermediateResult == 0)
41             {
42                 intermediateResult = _listComparer.Compare(left.Value, right.Value);
43             }
44             return intermediateResult;
45         }
46     }
47
48     public DuplicateSegmentsProvider(IList<TLink> links, ISequences<TLink> sequences)
    : base(minimumStringSegmentLength: 2)
49     {
50         _links = links;
51         _sequences = sequences;
52     }
53
54     public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
55     {
56         _groups = new HashSet<KeyValuePair<IList<TLink>,
    ↳ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
57         var count = _links.Count();
58         _visited = new BitString((long)(Integer<TLink>)count + 1);
59         _links.Each(link =>
60         {
61             var linkIndex = _links.GetIndex(link);
62             var linkBitIndex = (long)(Integer<TLink>)linkIndex;
63             if (!_visited.Get(linkBitIndex))
64             {
65                 var sequenceElements = new List<TLink>();
66                 _sequences.EachPart(sequenceElements.AddAndReturnTrue, linkIndex);
67                 if (sequenceElements.Count > 2)
68                 {
69                     WalkAll(sequenceElements);
70                 }
71             }
72             return _links.Constants.Continue;
73         });
74         var resultList = _groups.ToList();
75         var comparer = Default<ItemComparer>.Instance;
76         resultList.Sort(comparer);
77
78 #if DEBUG
79         foreach (var item in resultList)
80         {
81             PrintDuplicates(item);
82         }
83 #endif

```

```

84         return resultList;
85     }
86
87     protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
    ↪ length) => new Segment<TLink>(elements, offset, length);
88
89     protected override void OnDuplicateFound(Segment<TLink> segment)
90     {
91         var duplicates = CollectDuplicatesForSegment(segment);
92         if (duplicates.Count > 1)
93         {
94             _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
    ↪ duplicates));
95         }
96     }
97
98     private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
99     {
100         var duplicates = new List<TLink>();
101         var readAsElement = new HashSet<TLink>();
102         _sequences.Each(sequence =>
103         {
104             duplicates.Add(sequence);
105             readAsElement.Add(sequence);
106             return true; // Continue
107         }, segment);
108         if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
109         {
110             return new List<TLink>();
111         }
112         foreach (var duplicate in duplicates)
113         {
114             var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
115             _visited.Set(duplicateBitIndex);
116         }
117         if (_sequences is Sequences sequencesExperiments)
118         {
119             var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
    ↪
    ↪ ashSet<ulong>)(object)readAsElement,
    ↪ (IList<ulong>)segment);
120             foreach (var partiallyMatchedSequence in partiallyMatched)
121             {
122                 TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
123                 duplicates.Add(sequenceIndex);
124             }
125         }
126         duplicates.Sort();
127         return duplicates;
128     }
129
130     private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
131     {
132         if (!(_links is ILinks<ulong> ulongLinks))
133         {
134             return;
135         }
136         var duplicatesKey = duplicatesItem.Key;
137         var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
138         Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
139         var duplicatesList = duplicatesItem.Value;
140         for (int i = 0; i < duplicatesList.Count; i++)
141         {
142             ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
143             var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
    ↪
    ↪ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
    ↪ UnicodeMap.IsCharLink(link.Index) ?
    ↪ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
144             Console.WriteLine(formattedSequenceStructure);
145             var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
    ↪
    ↪ ulongLinks);
146             Console.WriteLine(sequenceString);
147         }
148         Console.WriteLine();
149     }
150 }
151 }

```

./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs

```
1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
5 {
6     public class FrequenciesCacheBasedLinkFrequencyIncrementer<TLink> :
7         ↳ IIncrementer<IList<TLink>>
8     {
9         private readonly LinkFrequenciesCache<TLink> _cache;
10
11         public FrequenciesCacheBasedLinkFrequencyIncrementer(LinkFrequenciesCache<TLink> cache)
12             ↳ => _cache = cache;
13
14         /// <remarks>Sequence itseft is not changed, only frequency of its doublets is
15         ↳ incremented.</remarks>
16         public IList<TLink> Increment(IList<TLink> sequence)
17         {
18             _cache.IncrementFrequencies(sequence);
19             return sequence;
20         }
21     }
22 }
```

./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToItsFrequencyNumberConverter.cs

```
1 using Platform.Interfaces;
2
3 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
4 {
5     public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
6         ↳ IConverter<Doublet<TLink>, TLink>
7     {
8         private readonly LinkFrequenciesCache<TLink> _cache;
9         public
10             ↳ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
11             ↳ cache) => _cache = cache;
12         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
13     }
14 }
```

./Sequences/Frequencies/Cache/LinkFrequenciesCache.cs

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

```

36 public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
37 {
38     _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
39     return data;
40 }
41
42 public void IncrementFrequencies(IList<TLink> sequence)
43 {
44     for (var i = 1; i < sequence.Count; i++)
45     {
46         IncrementFrequency(sequence[i - 1], sequence[i]);
47     }
48 }
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
52 {
53     var doublet = new Doublet<TLink>(source, target);
54     return IncrementFrequency(ref doublet);
55 }
56
57 public void PrintFrequencies(IList<TLink> sequence)
58 {
59     for (var i = 1; i < sequence.Count; i++)
60     {
61         PrintFrequency(sequence[i - 1], sequence[i]);
62     }
63 }
64
65 public void PrintFrequency(TLink source, TLink target)
66 {
67     var number = GetFrequency(source, target).Frequency;
68     Console.WriteLine("{0},{1}) - {2}", source, target, number);
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
73 {
74     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
75     {
76         data.IncrementFrequency();
77     }
78     else
79     {
80         var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
81         data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
82         if (!_equalityComparer.Equals(link, default))
83         {
84             data.Frequency = ArithmeticHelpers.Add(data.Frequency,
85                 ↪ _frequencyCounter.Count(link));
86         }
87         _doubletsCache.Add(doublet, data);
88     }
89     return data;
90 }
91
92 public void ValidateFrequencies()
93 {
94     foreach (var entry in _doubletsCache)
95     {
96         var value = entry.Value;
97         var linkIndex = value.Link;
98         if (!_equalityComparer.Equals(linkIndex, default))
99         {
100             var frequency = value.Frequency;
101             var count = _frequencyCounter.Count(linkIndex);
102             // TODO: Why `frequency` always greater than `count` by 1?
103             if (((_comparer.Compare(frequency, count) > 0) &&
104                 ↪ (_comparer.Compare(ArithmeticHelpers.Subtract(frequency, count),
105                 ↪ Integer<TLink>.One) > 0))
106                 || ((_comparer.Compare(count, frequency) > 0) &&
107                 ↪ (_comparer.Compare(ArithmeticHelpers.Subtract(count, frequency),
108                 ↪ Integer<TLink>.One) > 0)))
109             {
110                 throw new InvalidOperationException("Frequencies validation failed.");
111             }
112         }
113     }
114     //else

```



```

109         //{
110         //     if (value.Frequency > 0)
111         //     {
112         //         var frequency = value.Frequency;
113         //         linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
114         //         var count = _countLinkFrequency(linkIndex);
115
116         //         if ((frequency > count && frequency - count > 1) || (count > frequency
117         ↪      && count - frequency > 1))
118         //             throw new Exception("Frequencies validation failed.");
119         //     }
120         //}
121     }
122 }
123 }

```

./Sequences/Frequencies/Cache/LinkFrequency.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Numbers;
3
4  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
5  {
6      public class LinkFrequency<TLink>
7      {
8          public TLink Frequency { get; set; }
9          public TLink Link { get; set; }
10
11         public LinkFrequency(TLink frequency, TLink link)
12         {
13             Frequency = frequency;
14             Link = link;
15         }
16
17         public LinkFrequency() { }
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public void IncrementFrequency() => Frequency =
21         ↪      ArithmeticHelpers<TLink>.Increment(Frequency);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void DecrementFrequency() => Frequency =
25         ↪      ArithmeticHelpers<TLink>.Decrement(Frequency);
26
27         public override string ToString() => $"F: {Frequency}, L: {Link}";
28     }
29 }

```

./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs

```

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

```

./Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4  using Platform.Data.Sequences;

```

```

5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7 {
8     public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↳ EqualityComparer<TLink>.Default;
12         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13
14         protected readonly ILinks<TLink> _links;
15         protected readonly TLink _sequenceLink;
16         protected readonly TLink _symbol;
17         protected TLink _total;
18
19         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
20             ↳ TLink symbol)
21         {
22             _links = links;
23             _sequenceLink = sequenceLink;
24             _symbol = symbol;
25             _total = default;
26         }
27
28         public virtual TLink Count()
29         {
30             if (_comparer.Compare(_total, default) > 0)
31             {
32                 return _total;
33             }
34             StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
35                 ↳ IsElement, VisitElement);
36             return _total;
37         }
38
39         private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
40             ↳ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
41             ↳ 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 }

```

./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs

```

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

```

./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs

```

1 using Platform.Interfaces;
2 using Platform.Numbers;
3
4 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
5 {
6     public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
7         ↳ TotalSequenceSymbolFrequencyOneOffCounter<TLink>

```

```

7 {
8     private readonly ICriteriaMatcher<TLink> _markedSequenceMatcher;
9
10    public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
11        ↳ ICriteriaMatcher<TLink> markedSequenceMatcher, TLink symbol) : base(links, symbol)
12        => _markedSequenceMatcher = markedSequenceMatcher;
13
14    protected override void CountSequenceSymbolFrequency(TLink link)
15    {
16        var symbolFrequencyCounter = new
17            ↳ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
18            ↳ _markedSequenceMatcher, link, _symbol);
19        _total = ArithmeticHelpers.Add(_total, symbolFrequencyCounter.Count());
20    }
21 }

```

./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs

```

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

```

./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3 using Platform.Numbers;
4
5 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6 {
7     public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
8     {
9         private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↳ EqualityComparer<TLink>.Default;
11         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13         protected readonly ILinks<TLink> _links;
14         protected readonly TLink _symbol;
15         protected readonly HashSet<TLink> _visits;
16         protected TLink _total;
17
18         public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
19         {
20             _links = links;
21             _symbol = symbol;
22             _visits = new HashSet<TLink>();
23             _total = default;
24         }
25
26         public TLink Count()
27         {
28             if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
29             {
30                 return _total;
31             }
32             CountCore(_symbol);
33             return _total;
34         }
35
36         private void CountCore(TLink link)
37         {
38             var any = _links.Constants.Any;
39             if (_equalityComparer.Equals(_links.Count(any, link), default))
40             {
41                 CountSequenceSymbolFrequency(link);
42             }
43             else
44             {
45                 _links.Each(EachElementHandler, any, link);
46             }
47         }
48     }
49 }

```

```

48     protected virtual void CountSequenceSymbolFrequency(TLink link)
49     {
50         var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
51             ↪ link, _symbol);
52         _total = ArithmeticHelpers.Add(_total, symbolFrequencyCounter.Count());
53     }
54     private TLink EachElementHandler(IList<TLink> doublet)
55     {
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>,
7          ↪ ISequenceHeightProvider<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↪ EqualityComparer<TLink>.Default;
11
12         private readonly TLink _heightPropertyMarker;
13         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
14         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
15         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
16         private readonly IPropertyOperator<TLink, TLink, TLink> _propertyOperator;
17
18         public CachedSequenceHeightProvider(
19             ILinks<TLink> links,
20             ISequenceHeightProvider<TLink> baseHeightProvider,
21             IConverter<TLink> addressToUnaryNumberConverter,
22             IConverter<TLink> unaryNumberToAddressConverter,
23             TLink heightPropertyMarker,
24             IPropertyOperator<TLink, TLink, TLink> propertyOperator)
25             : base(links)
26         {
27             _heightPropertyMarker = heightPropertyMarker;
28             _baseHeightProvider = baseHeightProvider;
29             _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
30             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
31             _propertyOperator = propertyOperator;
32         }
33
34         public TLink Get(TLink sequence)
35         {
36             TLink height;
37             var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
38             if (_equalityComparer.Equals(heightValue, default))
39             {
40                 height = _baseHeightProvider.Get(sequence);
41                 heightValue = _addressToUnaryNumberConverter.Convert(height);
42                 _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
43             }
44             else
45             {
46                 height = _unaryNumberToAddressConverter.Convert(heightValue);
47             }
48             return height;
49         }
50     }
51 }

```

./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```

1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  namespace Platform.Data.Doublets.Sequences.HeightProviders
5  {
6      public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
7          ↪ ISequenceHeightProvider<TLink>

```

```

7 {
8     private readonly ICriteriaMatcher<TLink> _elementMatcher;
9
10    public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriteriaMatcher<TLink>
11        ↪ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
12
13    public TLink Get(TLink sequence)
14    {
15        var height = default(TLink);
16        var pairOrElement = sequence;
17        while (!_elementMatcher.IsMatched(pairOrElement))
18        {
19            pairOrElement = Links.GetTarget(pairOrElement);
20            height = ArithmeticHelpers.Increment(height);
21        }
22        return height;
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
26     ↪ вместе, все числа вместе и т.п.
27     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
28     ↪ графа)
29     ///
30     /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
31     ↪ ограничитель на то, что является последовательностью, а что нет,
32     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
33     ↪ порядке.
34     ///
35     /// Рост последовательности слева и справа.
36     /// Поиск со звездочкой.
37     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
38     /// так же проблема может быть решена при реализации дистанционных триггеров.
39     /// Нужны ли уникальные указатели вообще?
40     /// Что если обращение к информации будет происходить через содержимое всегда?
41     ///
42     /// Писать тесты.
43     ///
44     ///
45     /// Можно убрать зависимость от конкретной реализации Links,
46     /// на зависимость от абстрактного элемента, который может быть представлен несколькими
47     ↪ способами.
48     ///
49 }

```

```

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 Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
94          }
95          return sequence;
96      }
97
98      private ulong GetSequenceElements(ulong sequence)
99      {
100         if (Options.UseSequenceMarker)
101         {
102             var linkContents = new UInt64Link(Links.GetLink(sequence));
103             if (linkContents.Source == Options.SequenceMarkerLink)
104             {
105                 return linkContents.Target;
106             }
107             if (linkContents.Target == Options.SequenceMarkerLink)
108             {
109                 return linkContents.Source;
110             }
111         }
112         return sequence;
113     }
114
115     #region Count
116
117     public ulong Count(params ulong[] sequence)
118     {
119         if (sequence.Length == 0)
120         {

```

```

121         return Links.Count(_constants.Any, Options.SequenceMarkerLink, _constants.Any);
122     }
123     if (sequence.Length == 1) // Первая связь это адрес
124     {
125         if (sequence[0] == _constants.Null)
126         {
127             return 0;
128         }
129         if (sequence[0] == _constants.Any)
130         {
131             return Count();
132         }
133         if (Options.UseSequenceMarker)
134         {
135             return Links.Count(_constants.Any, Options.SequenceMarkerLink, sequence[0]);
136         }
137         return Links.Exists(sequence[0]) ? 1UL : 0;
138     }
139     throw new NotImplementedException();
140 }
141
142 private ulong CountReferences(params ulong[] restrictions)
143 {
144     if (restrictions.Length == 0)
145     {
146         return 0;
147     }
148     if (restrictions.Length == 1) // Первая связь это адрес
149     {
150         if (restrictions[0] == _constants.Null)
151         {
152             return 0;
153         }
154         if (Options.UseSequenceMarker)
155         {
156             var elementsLink = GetSequenceElements(restrictions[0]);
157             var sequenceLink = GetSequenceByElements(elementsLink);
158             if (sequenceLink != _constants.Null)
159             {
160                 return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
161             }
162             return Links.Count(elementsLink);
163         }
164         return Links.Count(restrictions[0]);
165     }
166     throw new NotImplementedException();
167 }
168
169 #endregion
170
171 #region Create
172
173 public ulong Create(params ulong[] sequence)
174 {
175     return Sync.ExecuteWriteOperation(() =>
176     {
177         if (sequence.IsNullOrEmpty())
178         {
179             return _constants.Null;
180         }
181         Links.EnsureEachLinkExists(sequence);
182         return CreateCore(sequence);
183     });
184 }
185
186 private ulong CreateCore(params ulong[] sequence)
187 {
188     if (Options.UseIndex)
189     {
190         Options.Indexer.Index(sequence);
191     }
192     var sequenceRoot = default(ulong);
193     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
194     {
195         var matches = Each(sequence);
196         if (matches.Count > 0)
197         {
198             sequenceRoot = matches[0];
199         }
200     }

```

```

200     }
201     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
202     {
203         return CompactCore(sequence);
204     }
205     if (sequenceRoot == default)
206     {
207         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
208     }
209     if (Options.UseSequenceMarker)
210     {
211         Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
212     }
213     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
214 }
215
216 #endregion
217
218 #region Each
219
220 public List<ulong> Each(params ulong[] sequence)
221 {
222     var results = new List<ulong>();
223     Each(results.AddAndReturnTrue, sequence);
224     return results;
225 }
226
227 public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
228 {
229     return Sync.ExecuteReadOperation(() =>
230     {
231         if (sequence.IsNullOrEmpty())
232         {
233             return true;
234         }
235         Links.EnsureEachLinkIsAnyOrExists(sequence);
236         if (sequence.Count == 1)
237         {
238             var link = sequence[0];
239             if (link == _constants.Any)
240             {
241                 return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
242             }
243             return handler(link);
244         }
245         if (sequence.Count == 2)
246         {
247             return Links.Unsync.Each(sequence[0], sequence[1], handler);
248         }
249         if (Options.UseIndex && !Options.Indexer.CheckIndex(sequence))
250         {
251             return false;
252         }
253         return EachCore(handler, sequence);
254     });
255 }
256
257 private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
258 {
259     var matcher = new Matcher(this, sequence, new HashSet<LinkIndex>(), handler);
260     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
261     // ↳ Id.
262     Func<ulong, bool> innerHandler = Options.UseSequenceMarker ? (Func<ulong,
263     // ↳ bool>)matcher.HandleFullMatchedSequence : matcher.HandleFullMatched;
264     //if (sequence.Length >= 2)
265     if (!StepRight(innerHandler, sequence[0], sequence[1]))
266     {
267         return false;
268     }
269     var last = sequence.Count - 2;
270     for (var i = 1; i < last; i++)
271     {
272         if (!PartialStepRight(innerHandler, sequence[i], sequence[i + 1]))
273         {
274             return false;
275         }
276     }
277     if (sequence.Count >= 3)
278     {

```



```

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

```

```

351     if (newSequence.IsNullOrEmpty())
352     {
353         Delete(sequence);
354         return _constants.Null;
355     }
356     return Sync.ExecuteWriteOperation(() =>
357     {
358         Links.EnsureEachLinkIsAnyOrExists(sequence);
359         Links.EnsureEachLinkExists(newSequence);
360         return UpdateCore(sequence, newSequence);
361     });
362 }
363
364 private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
365 {
366     ulong bestVariant;
367     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
368         ↪ !sequence.EqualTo(newSequence))
369     {
370         bestVariant = CompactCore(newSequence);
371     }
372     else
373     {
374         bestVariant = CreateCore(newSequence);
375     }
376     // TODO: Check all options only ones before loop execution
377     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
378     ↪ маркером,
379     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
380     ↪ можно получить имея только фактические последовательности.
381     foreach (var variant in Each(sequence))
382     {
383         if (variant != bestVariant)
384         {
385             UpdateOneCore(variant, bestVariant);
386         }
387     }
388     return bestVariant;
389 }
390
391 private void UpdateOneCore(ulong sequence, ulong newSequence)
392 {
393     if (Options.UseGarbageCollection)
394     {
395         var sequenceElements = GetSequenceElements(sequence);
396         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
397         var sequenceLink = GetSequenceByElements(sequenceElements);
398         var newSequenceElements = GetSequenceElements(newSequence);
399         var newSequenceLink = GetSequenceByElements(newSequenceElements);
400         if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
401         {
402             if (sequenceLink != _constants.Null)
403             {
404                 Links.Unsync.Merge(sequenceLink, newSequenceLink);
405             }
406             Links.Unsync.Merge(sequenceElements, newSequenceElements);
407         }
408         ClearGarbage(sequenceElementsContents.Source);
409         ClearGarbage(sequenceElementsContents.Target);
410     }
411     else
412     {
413         if (Options.UseSequenceMarker)
414         {
415             var sequenceElements = GetSequenceElements(sequence);
416             var sequenceLink = GetSequenceByElements(sequenceElements);
417             var newSequenceElements = GetSequenceElements(newSequence);
418             var newSequenceLink = GetSequenceByElements(newSequenceElements);
419             if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
420             {
421                 if (sequenceLink != _constants.Null)
422                 {
423                     Links.Unsync.Merge(sequenceLink, newSequenceLink);
424                 }
425                 Links.Unsync.Merge(sequenceElements, newSequenceElements);
426             }
427         }
428         else

```

```

426         {
427             if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
428             {
429                 Links.Unsync.Merge(sequence, newSequence);
430             }
431         }
432     }
433 }
434
435 #endregion
436
437 #region Delete
438
439 public void Delete(params ulong[] sequence)
440 {
441     Sync.ExecuteWriteOperation(() =>
442     {
443         // TODO: Check all options only ones before loop execution
444         foreach (var linkToDelete in Each(sequence))
445         {
446             DeleteOneCore(linkToDelete);
447         }
448     });
449 }
450
451 private void DeleteOneCore(ulong link)
452 {
453     if (Options.UseGarbageCollection)
454     {
455         var sequenceElements = GetSequenceElements(link);
456         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
457         var sequenceLink = GetSequenceByElements(sequenceElements);
458         if (Options.UseCascadeDelete || CountReferences(link) == 0)
459         {
460             if (sequenceLink != _constants.Null)
461             {
462                 Links.Unsync.Delete(sequenceLink);
463             }
464             Links.Unsync.Delete(link);
465         }
466         ClearGarbage(sequenceElementsContents.Source);
467         ClearGarbage(sequenceElementsContents.Target);
468     }
469     else
470     {
471         if (Options.UseSequenceMarker)
472         {
473             var sequenceElements = GetSequenceElements(link);
474             var sequenceLink = GetSequenceByElements(sequenceElements);
475             if (Options.UseCascadeDelete || CountReferences(link) == 0)
476             {
477                 if (sequenceLink != _constants.Null)
478                 {
479                     Links.Unsync.Delete(sequenceLink);
480                 }
481                 Links.Unsync.Delete(link);
482             }
483         }
484         else
485         {
486             if (Options.UseCascadeDelete || CountReferences(link) == 0)
487             {
488                 Links.Unsync.Delete(link);
489             }
490         }
491     }
492 }
493
494 #endregion
495
496 #region Compactification
497
498 /// <remarks>
499 /// bestVariant можно выбирать по максимальному числу использований,
500 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
501 /// гарантировать его использование в других местах).
502 ///
503 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
504 /// </remarks>

```

```

505 public ulong Compact(params ulong[] sequence)
506 {
507     return Sync.ExecuteWriteOperation(() =>
508     {
509         if (sequence.IsNullOrEmpty())
510         {
511             return _constants.Null;
512         }
513         Links.EnsureEachLinkExists(sequence);
514         return CompactCore(sequence);
515     });
516 }
517
518 [MethodImpl(MethodImplOptions.AggressiveInlining)]
519 private ulong CompactCore(params ulong[] sequence) => UpdateCore(sequence, sequence);
520
521 #endregion
522
523 #region Garbage Collection
524
525 /// <remarks>
526 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
527 /// ↳ определить извне или в унаследованном классе
528 /// </remarks>
529 [MethodImpl(MethodImplOptions.AggressiveInlining)]
530 private bool IsGarbage(ulong link) => link != Options.SequenceMarkerLink &&
531     ↳ !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;
532
533 private void ClearGarbage(ulong link)
534 {
535     if (IsGarbage(link))
536     {
537         var contents = new UInt64Link(Links.GetLink(link));
538         Links.Unsync.Delete(link);
539         ClearGarbage(contents.Source);
540         ClearGarbage(contents.Target);
541     }
542 }
543
544 #endregion
545
546 #region Walkers
547
548 public bool EachPart(Func<ulong, bool> handler, ulong sequence)
549 {
550     return Sync.ExecuteReadOperation(() =>
551     {
552         var links = Links.Unsync;
553         var walker = new RightSequenceWalker<ulong>(links);
554         foreach (var part in walker.Walk(sequence))
555         {
556             if (!handler(links.GetIndex(part)))
557             {
558                 return false;
559             }
560         }
561         return true;
562     });
563 }
564
565 public class Matcher : RightSequenceWalker<ulong>
566 {
567     private readonly Sequences _sequences;
568     private readonly IList<LinkIndex> _patternSequence;
569     private readonly HashSet<LinkIndex> _linksInSequence;
570     private readonly HashSet<LinkIndex> _results;
571     private readonly Func<ulong, bool> _stopableHandler;
572     private readonly HashSet<ulong> _readAsElements;
573     private int _filterPosition;
574
575     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
576         ↳ HashSet<LinkIndex> results, Func<LinkIndex, bool> stopableHandler,
577         ↳ HashSet<LinkIndex> readAsElements = null)
578         : base(sequences.Links.Unsync)
579     {
580         _sequences = sequences;
581         _patternSequence = patternSequence;
582         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
583             ↳ _constants.Any && x != ZeroOrMany));
584         _results = results;
585     }

```

```

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

```

./Sequences/Sequences.Experiments.cs

```

1 using System;
2 using LinkIndex = System.UInt64;

```

```

3 using System.Collections.Generic;
4 using Stack = System.Collections.Generic.Stack<ulong>;
5 using System.Linq;
6 using System.Text;
7 using Platform.Collections;
8 using Platform.Numbers;
9 using Platform.Data.Exceptions;
10 using Platform.Data.Sequences;
11 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
12 using Platform.Data.Doublets.Sequences.Walkers;
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     partial class Sequences
17     {
18         #region Create All Variants (Not Practical)
19
20         /// <remarks>
21         /// Number of links that is needed to generate all variants for
22         /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
23         /// </remarks>
24         public ulong[] CreateAllVariants2(ulong[] sequence)
25         {
26             return Sync.ExecuteWriteOperation(() =>
27             {
28                 if (sequence.IsNullOrEmpty())
29                 {
30                     return new ulong[0];
31                 }
32                 Links.EnsureEachLinkExists(sequence);
33                 if (sequence.Length == 1)
34                 {
35                     return sequence;
36                 }
37                 return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
38             });
39         }
40
41         private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
42         {
43             #if DEBUG
44                 if ((stopAt - startAt) < 0)
45                 {
46                     throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
47                         ↪ меньше или равен stopAt");
48                 }
49                 #endif
50                 if ((stopAt - startAt) == 0)
51                 {
52                     return new[] { sequence[startAt] };
53                 }
54                 if ((stopAt - startAt) == 1)
55                 {
56                     return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt], sequence[stopAt])
57                         ↪ };
58                 }
59                 var variants = new ulong[(ulong)MathHelpers.Catalan(stopAt - startAt)];
60                 var last = 0;
61                 for (var splitter = startAt; splitter < stopAt; splitter++)
62                 {
63                     var left = CreateAllVariants2Core(sequence, startAt, splitter);
64                     var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
65                     for (var i = 0; i < left.Length; i++)
66                     {
67                         for (var j = 0; j < right.Length; j++)
68                         {
69                             var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
70                             if (variant == _constants.Null)
71                             {
72                                 throw new NotImplementedException("Creation cancellation is not
73                                     ↪ implemented.");
74                             }
75                             variants[last++] = variant;
76                         }
77                     }
78                 }
79                 return variants;
80             }
81         }
82     }
83 }

```

```

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
105                 ↳ implemented.");
106         }
107         results.Add(link);
108         return results;
109     }
110     var innerSequenceLength = sequence.Length - 1;
111     var innerSequence = new ulong[innerSequenceLength];
112     for (var li = 0; li < innerSequenceLength; li++)
113     {
114         var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
115         if (link == _constants.Null)
116         {
117             throw new NotImplementedException("Creation cancellation is not
118                 ↳ implemented.");
119         }
120         for (var isi = 0; isi < li; isi++)
121         {
122             innerSequence[isi] = sequence[isi];
123         }
124         innerSequence[li] = link;
125         for (var isi = li + 1; isi < innerSequenceLength; isi++)
126         {
127             innerSequence[isi] = sequence[isi + 1];
128         }
129         CreateAllVariants1Core(innerSequence, results);
130     }
131     return results;
132 }
133
134 #endregion
135
136 public HashSet<ulong> Each1(params ulong[] sequence)
137 {
138     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
139     Each1(link =>
140     {
141         if (!visitedLinks.Contains(link))
142         {
143             visitedLinks.Add(link); // изучить почему случаются повторы
144         }
145         return true;
146     }, sequence);
147     return visitedLinks;
148 }
149
150 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
151 {
152     if (sequence.Length == 2)
153     {
154         Links.Unsync.Each(sequence[0], sequence[1], handler);
155     }
156     else

```



```

155     {
156         var innerSequenceLength = sequence.Length - 1;
157         for (var li = 0; li < innerSequenceLength; li++)
158         {
159             var left = sequence[li];
160             var right = sequence[li + 1];
161             if (left == 0 && right == 0)
162             {
163                 continue;
164             }
165             var linkIndex = li;
166             ulong[] innerSequence = null;
167             Links.Unsync.Each(left, right, doublet =>
168             {
169                 if (innerSequence == null)
170                 {
171                     innerSequence = new ulong[innerSequenceLength];
172                     for (var isi = 0; isi < linkIndex; isi++)
173                     {
174                         innerSequence[isi] = sequence[isi];
175                     }
176                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
177                     {
178                         innerSequence[isi] = sequence[isi + 1];
179                     }
180                 }
181                 innerSequence[linkIndex] = doublet;
182                 Each1(handler, innerSequence);
183                 return _constants.Continue;
184             });
185         }
186     }
187 }
188
189 public HashSet<ulong> EachPart(params ulong[] sequence)
190 {
191     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
192     EachPartCore(link =>
193     {
194         if (!visitedLinks.Contains(link))
195         {
196             visitedLinks.Add(link); // изучить почему случаются повторы
197         }
198         return true;
199     }, sequence);
200     return visitedLinks;
201 }
202
203 public void EachPart(Func<ulong, bool> handler, params ulong[] sequence)
204 {
205     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
206     EachPartCore(link =>
207     {
208         if (!visitedLinks.Contains(link))
209         {
210             visitedLinks.Add(link); // изучить почему случаются повторы
211             return handler(link);
212         }
213         return true;
214     }, sequence);
215 }
216
217 private void EachPartCore(Func<ulong, bool> handler, params ulong[] sequence)
218 {
219     if (sequence.IsNullOrEmpty())
220     {
221         return;
222     }
223     Links.EnsureEachLinkIsAnyOrExists(sequence);
224     if (sequence.Length == 1)
225     {
226         var link = sequence[0];
227         if (link > 0)
228         {
229             handler(link);
230         }
231         else
232         {
233             Links.Each(_constants.Any, _constants.Any, handler);

```

```

234     }
235 }
236 else if (sequence.Length == 2)
237 {
238     //_links.Each(sequence[0], sequence[1], handler);
239     //  o_|      x_o ...
240     // x_|      |___|
241     Links.Each(sequence[1], _constants.Any, doublet =>
242     {
243         var match = Links.SearchOrDefault(sequence[0], doublet);
244         if (match != _constants.Null)
245         {
246             handler(match);
247         }
248         return true;
249     });
250     // |_x      ... x_o
251     // |_o      |___|
252     Links.Each(_constants.Any, sequence[0], doublet =>
253     {
254         var match = Links.SearchOrDefault(doublet, sequence[1]);
255         if (match != 0)
256         {
257             handler(match);
258         }
259         return true;
260     });
261     //          .x o_
262     //          |___|
263     PartialStepRight(x => handler(x), sequence[0], sequence[1]);
264 }
265 else
266 {
267     // TODO: Implement other variants
268     return;
269 }
270 }
271
272 private void PartialStepRight(Action<ulong> handler, ulong left, ulong right)
273 {
274     Links.Unsync.Each(_constants.Any, left, doublet =>
275     {
276         StepRight(handler, doublet, right);
277         if (left != doublet)
278         {
279             PartialStepRight(handler, doublet, right);
280         }
281         return true;
282     });
283 }
284
285 private void StepRight(Action<ulong> handler, ulong left, ulong right)
286 {
287     Links.Unsync.Each(left, _constants.Any, rightStep =>
288     {
289         TryStepRightUp(handler, right, rightStep);
290         return true;
291     });
292 }
293
294 private void TryStepRightUp(Action<ulong> handler, ulong right, ulong stepFrom)
295 {
296     var upStep = stepFrom;
297     var firstSource = Links.Unsync.GetTarget(upStep);
298     while (firstSource != right && firstSource != upStep)
299     {
300         upStep = firstSource;
301         firstSource = Links.Unsync.GetSource(upStep);
302     }
303     if (firstSource == right)
304     {
305         handler(stepFrom);
306     }
307 }
308
309 // TODO: Test
310 private void PartialStepLeft(Action<ulong> handler, ulong left, ulong right)
311 {
312     Links.Unsync.Each(right, _constants.Any, doublet =>

```

```

313     {
314         StepLeft(handler, left, doublet);
315         if (right != doublet)
316         {
317             PartialStepLeft(handler, left, doublet);
318         }
319         return true;
320     });
321 }
322
323 private void StepLeft(Action<ulong> handler, ulong left, ulong right)
324 {
325     Links.Unsync.Each(_constants.Any, right, leftStep =>
326     {
327         TryStepLeftUp(handler, left, leftStep);
328         return true;
329     });
330 }
331
332 private void TryStepLeftUp(Action<ulong> handler, ulong left, ulong stepFrom)
333 {
334     var upStep = stepFrom;
335     var firstTarget = Links.Unsync.GetSource(upStep);
336     while (firstTarget != left && firstTarget != upStep)
337     {
338         upStep = firstTarget;
339         firstTarget = Links.Unsync.GetTarget(upStep);
340     }
341     if (firstTarget == left)
342     {
343         handler(stepFrom);
344     }
345 }
346
347 private bool StartsWith(ulong sequence, ulong link)
348 {
349     var upStep = sequence;
350     var firstSource = Links.Unsync.GetSource(upStep);
351     while (firstSource != link && firstSource != upStep)
352     {
353         upStep = firstSource;
354         firstSource = Links.Unsync.GetSource(upStep);
355     }
356     return firstSource == link;
357 }
358
359 private bool EndsWith(ulong sequence, ulong link)
360 {
361     var upStep = sequence;
362     var lastTarget = Links.Unsync.GetTarget(upStep);
363     while (lastTarget != link && lastTarget != upStep)
364     {
365         upStep = lastTarget;
366         lastTarget = Links.Unsync.GetTarget(upStep);
367     }
368     return lastTarget == link;
369 }
370
371 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
372 {
373     return Sync.ExecuteReadOperation(() =>
374     {
375         var results = new List<ulong>();
376         if (sequence.Length > 0)
377         {
378             Links.EnsureEachLinkExists(sequence);
379             var firstElement = sequence[0];
380             if (sequence.Length == 1)
381             {
382                 results.Add(firstElement);
383                 return results;
384             }
385             if (sequence.Length == 2)
386             {
387                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
388                 if (doublet != _constants.Null)
389                 {
390                     results.Add(doublet);
391                 }

```

```

392         return results;
393     }
394     var linksInSequence = new HashSet<ulong>(sequence);
395     void handler(ulong result)
396     {
397         var filterPosition = 0;
398         StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
399             ↪ Links.Unsync.GetTarget,
400             x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
401             ↪ x =>
402             {
403                 if (filterPosition == sequence.Length)
404                 {
405                     filterPosition = -2; // Длиннее чем нужно
406                     return false;
407                 }
408                 if (x != sequence[filterPosition])
409                 {
410                     filterPosition = -1;
411                     return false; // Начинается иначе
412                 }
413                 filterPosition++;
414                 return true;
415             });
416         if (filterPosition == sequence.Length)
417         {
418             results.Add(result);
419         }
420     }
421     if (sequence.Length >= 2)
422     {
423         StepRight(handler, sequence[0], sequence[1]);
424     }
425     var last = sequence.Length - 2;
426     for (var i = 1; i < last; i++)
427     {
428         PartialStepRight(handler, sequence[i], sequence[i + 1]);
429     }
430     if (sequence.Length >= 3)
431     {
432         StepLeft(handler, sequence[sequence.Length - 2],
433             ↪ sequence[sequence.Length - 1]);
434     }
435     }
436     return results;
437 }
438 });
439
440 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
441 {
442     return Sync.ExecuteReadOperation(() =>
443     {
444         var results = new HashSet<ulong>();
445         if (sequence.Length > 0)
446         {
447             Links.EnsureEachLinkExists(sequence);
448             var firstElement = sequence[0];
449             if (sequence.Length == 1)
450             {
451                 results.Add(firstElement);
452                 return results;
453             }
454             if (sequence.Length == 2)
455             {
456                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
457                 if (doublet != _constants.Null)
458                 {
459                     results.Add(doublet);
460                 }
461                 return results;
462             }
463             var matcher = new Matcher(this, sequence, results, null);
464             if (sequence.Length >= 2)
465             {
466                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
467             }
468             var last = sequence.Length - 2;
469             for (var i = 1; i < last; i++)

```

```

468         {
469             PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
470                             ↪ sequence[i + 1]);
471         }
472         if (sequence.Length >= 3)
473         {
474             StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
475                     ↪ sequence[sequence.Length - 1]);
476         }
477     }
478     return results;
479 }
480 }
481
482 public const int MaxSequenceFormatSize = 200;
483
484 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
485     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
486
487 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
488     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
489     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
490     ↪ elementToString, insertComma, knownElements));
491
492 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
493     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
494     ↪ LinkIndex[] knownElements)
495 {
496     var linksInSequence = new HashSet<ulong>(knownElements);
497     //var entered = new HashSet<ulong>();
498     var sb = new StringBuilder();
499     sb.Append('{');
500     if (links.Exists(sequenceLink))
501     {
502         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
503             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
504             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
505         {
506             if (insertComma && sb.Length > 1)
507             {
508                 sb.Append(',');
509             }
510             //if (entered.Contains(element))
511             //{
512             //    sb.Append('{');
513             //    elementToString(sb, element);
514             //    sb.Append('}');
515             //}
516             //else
517             elementToString(sb, element);
518             if (sb.Length < MaxSequenceFormatSize)
519             {
520                 return true;
521             }
522             sb.Append(insertComma ? ", ..." : "...");
523             return false;
524         }
525     }
526     sb.Append('}');
527     return sb.ToString();
528 }
529
530 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
531     ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
532     ↪ knownElements);
533
534 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
535     ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
536     ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
537     ↪ sequenceLink, elementToString, insertComma, knownElements));
538
539 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
540     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
541     ↪ LinkIndex[] knownElements)
542 {
543     var linksInSequence = new HashSet<ulong>(knownElements);
544     var entered = new HashSet<ulong>();

```

```

529     var sb = new StringBuilder();
530     sb.Append('{');
531     if (links.Exists(sequenceLink))
532     {
533         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
534             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
535             entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
536             {
537                 if (insertComma && sb.Length > 1)
538                 {
539                     sb.Append(',');
540                 }
541                 if (entered.Contains(element))
542                 {
543                     sb.Append('{');
544                     elementToString(sb, element);
545                     sb.Append('}');
546                 }
547                 else
548                 {
549                     elementToString(sb, element);
550                 }
551                 if (sb.Length < MaxSequenceFormatSize)
552                 {
553                     return true;
554                 }
555                 sb.Append(insertComma ? ", ..." : "...");
556                 return false;
557             });
558     }
559     sb.Append('}');
560     return sb.ToString();
561 }
562 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
563 {
564     return Sync.ExecuteReadOperation(() =>
565     {
566         if (sequence.Length > 0)
567         {
568             Links.EnsureEachLinkExists(sequence);
569             var results = new HashSet<ulong>();
570             for (var i = 0; i < sequence.Length; i++)
571             {
572                 AllUsagesCore(sequence[i], results);
573             }
574             var filteredResults = new List<ulong>();
575             var linksInSequence = new HashSet<ulong>(sequence);
576             foreach (var result in results)
577             {
578                 var filterPosition = -1;
579                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
580                     Links.Unsync.GetTarget,
581                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
582                     x =>
583                     {
584                         if (filterPosition == (sequence.Length - 1))
585                         {
586                             return false;
587                         }
588                         if (filterPosition >= 0)
589                         {
590                             if (x == sequence[filterPosition + 1])
591                             {
592                                 filterPosition++;
593                             }
594                             else
595                             {
596                                 return false;
597                             }
598                         }
599                         if (filterPosition < 0)
600                         {
601                             if (x == sequence[0])
602                             {
603                                 filterPosition = 0;
604                             }
605                         }
606                     }
607             }
608         }
609     });
610 }

```

```

604         return true;
605     });
606     if (filterPosition == (sequence.Length - 1))
607     {
608         filteredResults.Add(result);
609     }
610 }
611 return filteredResults;
612 }
613 return new List<ulong>();
614 });
615 }
616
617 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
618 {
619     return Sync.ExecuteReadOperation(() =>
620     {
621         if (sequence.Length > 0)
622         {
623             Links.EnsureEachLinkExists(sequence);
624             var results = new HashSet<ulong>();
625             for (var i = 0; i < sequence.Length; i++)
626             {
627                 AllUsagesCore(sequence[i], results);
628             }
629             var filteredResults = new HashSet<ulong>();
630             var matcher = new Matcher(this, sequence, filteredResults, null);
631             matcher.AddAllPartialMatchedToResults(results);
632             return filteredResults;
633         }
634         return new HashSet<ulong>();
635     });
636 }
637
638 public bool GetAllPartiallyMatchingSequences2(Func<ulong, bool> handler, params ulong[]
639 → 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, filteredResults, handler);
650             for (var i = 0; i < sequence.Length; i++)
651             {
652                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
653                 {
654                     return false;
655                 }
656             }
657             return true;
658         }
659         return true;
660     });
661 }
662
663 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
664 //{
665 //    return Sync.ExecuteReadOperation(() =>
666 //    {
667 //        if (sequence.Length > 0)
668 //        {
669 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
670 //
671 //            var firstResults = new HashSet<ulong>();
672 //            var lastResults = new HashSet<ulong>();
673 //
674 //            var first = sequence.First(x => x != LinksConstants.Any);
675 //            var last = sequence.Last(x => x != LinksConstants.Any);
676 //
677 //            AllUsagesCore(first, firstResults);
678 //            AllUsagesCore(last, lastResults);
679 //
680 //            firstResults.IntersectWith(lastResults);
681 //
682 //            //for (var i = 0; i < sequence.Length; i++)

```

```

682         //         // AllUsagesCore(sequence[i], results);
683
684         //         var filteredResults = new HashSet<ulong>();
685         //         var matcher = new Matcher(this, sequence, filteredResults, null);
686         //         matcher.AddAllPartialMatchedToResults(firstResults);
687         //         return filteredResults;
688         //     }
689
690     //     return new HashSet<ulong>();
691     // });
692 //}
693
694 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
695 {
696     return Sync.ExecuteReadOperation(() =>
697     {
698         if (sequence.Length > 0)
699         {
700             Links.EnsureEachLinkIsAnyOrExists(sequence);
701             var firstResults = new HashSet<ulong>();
702             var lastResults = new HashSet<ulong>();
703             var first = sequence.First(x => x != _constants.Any);
704             var last = sequence.Last(x => x != _constants.Any);
705             AllUsagesCore(first, firstResults);
706             AllUsagesCore(last, lastResults);
707             firstResults.IntersectWith(lastResults);
708             //for (var i = 0; i < sequence.Length; i++)
709             //    AllUsagesCore(sequence[i], results);
710             var filteredResults = new HashSet<ulong>();
711             var matcher = new Matcher(this, sequence, filteredResults, null);
712             matcher.AddAllPartialMatchedToResults(firstResults);
713             return filteredResults;
714         }
715         return new HashSet<ulong>();
716     });
717 }
718
719 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
720     → IList<ulong> sequence)
721 {
722     return Sync.ExecuteReadOperation(() =>
723     {
724         if (sequence.Count > 0)
725         {
726             Links.EnsureEachLinkExists(sequence);
727             var results = new HashSet<LinkIndex>();
728             //var nextResults = new HashSet<ulong>();
729             //for (var i = 0; i < sequence.Length; i++)
730             //{
731             //    AllUsagesCore(sequence[i], nextResults);
732             //    if (results.IsNullOrEmpty())
733             //    {
734             //        results = nextResults;
735             //        nextResults = new HashSet<ulong>();
736             //    }
737             //    else
738             //    {
739             //        results.IntersectWith(nextResults);
740             //        nextResults.Clear();
741             //    }
742             //}
743             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
744             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
745             var next = new HashSet<ulong>();
746             for (var i = 1; i < sequence.Count; i++)
747             {
748                 var collector = new AllUsagesCollector1(Links.Unsync, next);
749                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
750
751                 results.IntersectWith(next);
752                 next.Clear();
753             }
754             var filteredResults = new HashSet<ulong>();
755             var matcher = new Matcher(this, sequence, filteredResults, null,
756                 → readAsElements);
757             matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
758                 → x)); // OrderBy is a Hack
759             return filteredResults;

```



```

    }
    return new HashSet<ulong>();
});
}

// Does not work
public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
    ↪ params ulong[] sequence)
{
    var visited = new HashSet<ulong>();
    var results = new HashSet<ulong>();
    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
    ↪ true; }, readAsElements);
    var last = sequence.Length - 1;
    for (var i = 0; i < last; i++)
    {
        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
    }
    return results;
}

public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
{
    return Sync.ExecuteReadOperation(() =>
    {
        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.SearchCore(firstElement, sequence[1]);
            //    //if (doublet != Doublets.Links.Null)
            //    //    results.Add(doublet);
            //    return results;
            //}
            //var lastElement = sequence[sequence.Length - 1];
            //Func<ulong, bool> handler = x =>
            //{
            //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
            //        ↪ results.Add(x);
            //    return true;
            //};
            //if (sequence.Length >= 2)
            //    StepRight(handler, sequence[0], sequence[1]);
            //var last = sequence.Length - 2;
            //for (var i = 1; i < last; i++)
            //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
            //if (sequence.Length >= 3)
            //    StepLeft(handler, sequence[sequence.Length - 2],
            //        ↪ sequence[sequence.Length - 1]);
            //if (sequence.Length == 1)
            //    throw new NotImplementedException(); // all sequences, containing
            //    ↪ this element?
            //if (sequence.Length == 2)
            //{
            //    var results = new List<ulong>();
            //    PartialStepRight(results.Add, sequence[0], sequence[1]);
            //    return results;
            //}
            //var matches = new List<List<ulong>>();
            //var last = sequence.Length - 1;
            //for (var i = 0; i < last; i++)
            //{
            //    var results = new List<ulong>();
            //    //StepRight(results.Add, sequence[i], sequence[i + 1]);
            //    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
            //    if (results.Count > 0)
            //        matches.Add(results);
            //    else

```

```

829         return results;
830         if (matches.Count == 2)
831         {
832             var merged = new List<ulong>();
833             for (var j = 0; j < matches[0].Count; j++)
834                 for (var k = 0; k < matches[1].Count; k++)
835                     CloseInnerConnections(merged.Add, matches[0][j],
836                                     ↪ matches[1][k]);
837             if (merged.Count > 0)
838                 matches = new List<List<ulong>> { merged };
839             else
840                 return new List<ulong>();
841         }
842         if (matches.Count > 0)
843         {
844             var usages = new HashSet<ulong>();
845             for (int i = 0; i < sequence.Length; i++)
846             {
847                 AllUsagesCore(sequence[i], usages);
848             }
849             //for (int i = 0; i < matches[0].Count; i++)
850             //    AllUsagesCore(matches[0][i], usages);
851             //usages.UnionWith(matches[0]);
852             return usages.ToList();
853         }
854         var firstLinkUsages = new HashSet<ulong>();
855         AllUsagesCore(sequence[0], firstLinkUsages);
856         firstLinkUsages.Add(sequence[0]);
857         //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
858         ↪ sequence[0] }; // or all sequences, containing this element?
859         //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
860         ↪ 1).ToList();
861         var results = new HashSet<ulong>();
862         foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
863         ↪ firstLinkUsages, 1))
864         {
865             AllUsagesCore(match, results);
866         }
867         return results.ToList();
868     }
869     return new List<ulong>();
870 });
871 }
872
873 /// <remarks>
874 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
875 /// </remarks>
876 public HashSet<ulong> AllUsages(ulong link)
877 {
878     return Sync.ExecuteReadOperation(() =>
879     {
880         var usages = new HashSet<ulong>();
881         AllUsagesCore(link, usages);
882         return usages;
883     });
884 }
885
886 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
887 ↪ той связи с которой начинался поиск (STTTSSSTT),
888 // причём достаточно одного бита для хранения перехода влево 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 {

```

```

901     return Sync.ExecuteReadOperation(() =>
902     {
903         var visits = new HashSet<ulong>();
904         var usages = new HashSet<ulong>();
905         AllBottomUsagesCore(link, visits, usages);
906         return usages;
907     });
908 }
909
910 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
↪ usages)
911 {
912     bool handler(ulong doublet)
913     {
914         if (visits.Add(doublet))
915         {
916             AllBottomUsagesCore(doublet, visits, usages);
917         }
918         return true;
919     }
920     if (Links.Unsync.Count(_constants.Any, link) == 0)
921     {
922         usages.Add(link);
923     }
924     else
925     {
926         Links.Unsync.Each(link, _constants.Any, handler);
927         Links.Unsync.Each(_constants.Any, link, handler);
928     }
929 }
930
931 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
932 {
933     if (Options.UseSequenceMarker)
934     {
935         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
↪ Options.MarkedSequenceMatcher, symbol);
936         return counter.Count();
937     }
938     else
939     {
940         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
↪ symbol);
941         return counter.Count();
942     }
943 }
944
945 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool>
↪ outerHandler)
946 {
947     bool handler(ulong doublet)
948     {
949         if (usages.Add(doublet))
950         {
951             if (!outerHandler(doublet))
952             {
953                 return false;
954             }
955             if (!AllUsagesCore1(doublet, usages, outerHandler))
956             {
957                 return false;
958             }
959         }
960         return true;
961     }
962     return Links.Unsync.Each(link, _constants.Any, handler)
963         && Links.Unsync.Each(_constants.Any, link, handler);
964 }
965
966 public void CalculateAllUsages(ulong[] totals)
967 {
968     var calculator = new AllUsagesCalculator(Links, totals);
969     calculator.Calculate();
970 }
971
972 public void CalculateAllUsages2(ulong[] totals)
973 {
974     var calculator = new AllUsagesCalculator2(Links, totals);

```

```

975     calculator.Calculate();
976 }
977
978 private class AllUsagesCalculator
979 {
980     private readonly SynchronizedLinks<ulong> _links;
981     private readonly ulong[] _totals;
982
983     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
984     {
985         _links = links;
986         _totals = totals;
987     }
988
989     public void Calculate() => _links.Each(_constants.Any, _constants.Any,
990         ↪ CalculateCore);
991
992     private bool CalculateCore(ulong link)
993     {
994         if (_totals[link] == 0)
995         {
996             var total = 1UL;
997             _totals[link] = total;
998             var visitedChildren = new HashSet<ulong>();
999             bool linkCalculator(ulong child)
1000             {
1001                 if (link != child && visitedChildren.Add(child))
1002                 {
1003                     total += _totals[child] == 0 ? 1 : _totals[child];
1004                 }
1005                 return true;
1006             }
1007             _links.Unsync.Each(link, _constants.Any, linkCalculator);
1008             _links.Unsync.Each(_constants.Any, link, linkCalculator);
1009             _totals[link] = total;
1010         }
1011         return true;
1012     }
1013 }
1014
1015 private class AllUsagesCalculator2
1016 {
1017     private readonly SynchronizedLinks<ulong> _links;
1018     private readonly ulong[] _totals;
1019
1020     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1021     {
1022         _links = links;
1023         _totals = totals;
1024     }
1025
1026     public void Calculate() => _links.Each(_constants.Any, _constants.Any,
1027         ↪ CalculateCore);
1028
1029     private bool IsElement(ulong link)
1030     {
1031         // _linksInSequence.Contains(link) ||
1032         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1033             ↪ link;
1034     }
1035
1036     private bool CalculateCore(ulong link)
1037     {
1038         // TODO: Проработать защиту от заикливания
1039         // Основано на SequenceWalker.WalkLeft
1040         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1041         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1042         Func<ulong, bool> isElement = IsElement;
1043         void visitLeaf(ulong parent)
1044         {
1045             if (link != parent)
1046             {
1047                 _totals[parent]++;
1048             }
1049         }
1050         void visitNode(ulong parent)
1051         {
1052             if (link != parent)
1053             {

```

```

1051         _totals[parent]++;
1052     }
1053 }
1054 var stack = new Stack();
1055 var element = link;
1056 if (isElement(element))
1057 {
1058     visitLeaf(element);
1059 }
1060 else
1061 {
1062     while (true)
1063     {
1064         if (isElement(element))
1065         {
1066             if (stack.Count == 0)
1067             {
1068                 break;
1069             }
1070             element = stack.Pop();
1071             var source = getSource(element);
1072             var target = getTarget(element);
1073             // 06пабoтка элeмeнтa
1074             if (isElement(target))
1075             {
1076                 visitLeaf(target);
1077             }
1078             if (isElement(source))
1079             {
1080                 visitLeaf(source);
1081             }
1082             element = source;
1083         }
1084         else
1085         {
1086             stack.Push(element);
1087             visitNode(element);
1088             element = getTarget(element);
1089         }
1090     }
1091     _totals[link]++;
1092     return true;
1093 }
1094 }
1095 }
1096
1097 private class AllUsagesCollector
1098 {
1099     private readonly ILinks<ulong> _links;
1100     private readonly HashSet<ulong> _usages;
1101
1102     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1103     {
1104         _links = links;
1105         _usages = usages;
1106     }
1107
1108     public bool Collect(ulong link)
1109     {
1110         if (_usages.Add(link))
1111         {
1112             _links.Each(link, _constants.Any, Collect);
1113             _links.Each(_constants.Any, link, Collect);
1114         }
1115         return true;
1116     }
1117 }
1118
1119 private class AllUsagesCollector1
1120 {
1121     private readonly ILinks<ulong> _links;
1122     private readonly HashSet<ulong> _usages;
1123     private readonly ulong _continue;
1124
1125     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1126     {
1127         _links = links;
1128         _usages = usages;
1129         _continue = _links.Constants.Continue;
1130     }

```

```

1131     public ulong Collect(IList<ulong> link)
1132     {
1133         var linkIndex = _links.GetIndex(link);
1134         if (_usages.Add(linkIndex))
1135         {
1136             _links.Each(Collect, _constants.Any, linkIndex);
1137         }
1138         return _continue;
1139     }
1140 }
1141
1142 private class AllUsagesCollector2
1143 {
1144     private readonly ILinks<ulong> _links;
1145     private readonly BitString _usages;
1146
1147     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1148     {
1149         _links = links;
1150         _usages = usages;
1151     }
1152
1153     public bool Collect(ulong link)
1154     {
1155         if (_usages.Add((long)link))
1156         {
1157             _links.Each(link, _constants.Any, Collect);
1158             _links.Each(_constants.Any, link, Collect);
1159         }
1160         return true;
1161     }
1162 }
1163
1164 private class AllUsagesIntersectingCollector
1165 {
1166     private readonly SynchronizedLinks<ulong> _links;
1167     private readonly HashSet<ulong> _intersectWith;
1168     private readonly HashSet<ulong> _usages;
1169     private readonly HashSet<ulong> _enter;
1170
1171     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
1172 ↪ intersectWith, HashSet<ulong> usages)
1173     {
1174         _links = links;
1175         _intersectWith = intersectWith;
1176         _usages = usages;
1177         _enter = new HashSet<ulong>(); // защита от заикливания
1178     }
1179
1180     public bool Collect(ulong link)
1181     {
1182         if (_enter.Add(link))
1183         {
1184             if (_intersectWith.Contains(link))
1185             {
1186                 _usages.Add(link);
1187             }
1188             _links.Unsync.Each(link, _constants.Any, Collect);
1189             _links.Unsync.Each(_constants.Any, link, Collect);
1190         }
1191         return true;
1192     }
1193 }
1194
1195 private void CloseInnerConnections(Action<ulong> handler, ulong left, ulong right)
1196 {
1197     TryStepLeftUp(handler, left, right);
1198     TryStepRightUp(handler, right, left);
1199 }
1200
1201 private void AllCloseConnections(Action<ulong> handler, ulong left, ulong right)
1202 {
1203     // Direct
1204     if (left == right)
1205     {
1206         handler(left);
1207     }
1208     var doublet = Links.Unsync.SearchOrDefault(left, right);
1209     if (doublet != _constants.Null)

```

```

1210     {
1211         handler(doublet);
1212     }
1213     // Inner
1214     CloseInnerConnections(handler, left, right);
1215     // Outer
1216     StepLeft(handler, left, right);
1217     StepRight(handler, left, right);
1218     PartialStepRight(handler, left, right);
1219     PartialStepLeft(handler, left, right);
1220 }
1221
1222 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
1223     ↳ HashSet<ulong> previousMatchings, long startAt)
1224 {
1225     if (startAt >= sequence.Length) // ?
1226     {
1227         return previousMatchings;
1228     }
1229     var secondLinkUsages = new HashSet<ulong>();
1230     AllUsagesCore(sequence[startAt], secondLinkUsages);
1231     secondLinkUsages.Add(sequence[startAt]);
1232     var matchings = new HashSet<ulong>();
1233     //for (var i = 0; i < previousMatchings.Count; i++)
1234     foreach (var secondLinkUsage in secondLinkUsages)
1235     {
1236         foreach (var previousMatching in previousMatchings)
1237         {
1238             //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1239             ↳ secondLinkUsage);
1240             StepRight(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
1241             TryStepRightUp(matchings.AddAndReturnVoid, secondLinkUsage,
1242             ↳ previousMatching);
1243             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1244             ↳ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1245             ↳ желаемым результатам.
1246             PartialStepRight(matchings.AddAndReturnVoid, previousMatching,
1247             ↳ secondLinkUsage);
1248         }
1249     }
1250     if (matchings.Count == 0)
1251     {
1252         return matchings;
1253     }
1254     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1255 }
1256
1257 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
1258     ↳ links, params ulong[] sequence)
1259 {
1260     if (sequence == null)
1261     {
1262         return;
1263     }
1264     for (var i = 0; i < sequence.Length; i++)
1265     {
1266         if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany &&
1267             ↳ !links.Exists(sequence[i]))
1268         {
1269             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1270             ↳ $"patternSequence[{i}]");
1271         }
1272     }
1273 }
1274
1275 // Pattern Matching -> Key To Triggers
1276 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1277 {
1278     return Sync.ExecuteReadOperation(() =>
1279     {
1280         patternSequence = Simplify(patternSequence);
1281         if (patternSequence.Length > 0)
1282         {
1283             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1284             var uniqueSequenceElements = new HashSet<ulong>();
1285             for (var i = 0; i < patternSequence.Length; i++)
1286             {

```

```

1278         if (patternSequence[i] != _constants.Any && patternSequence[i] !=
1279             ↪ ZeroOrMany)
1280         {
1281             uniqueSequenceElements.Add(patternSequence[i]);
1282         }
1283     }
1284     var results = new HashSet<ulong>();
1285     foreach (var uniqueSequenceElement in uniqueSequenceElements)
1286     {
1287         AllUsagesCore(uniqueSequenceElement, results);
1288     }
1289     var filteredResults = new HashSet<ulong>();
1290     var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1291     matcher.AddAllPatternMatchedToResults(results);
1292     return filteredResults;
1293 }
1294 return new HashSet<ulong>();
1295 });
1296 }
1297 // Найти все возможные связи между указанным списком связей.
1298 // Находит связи между всеми указанными связями в любом порядке.
1299 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1300 ↪ несколько раз в последовательности)
1301 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1302 {
1303     return Sync.ExecuteReadOperation(() =>
1304     {
1305         var results = new HashSet<ulong>();
1306         if (linksToConnect.Length > 0)
1307         {
1308             Links.EnsureEachLinkExists(linksToConnect);
1309             AllUsagesCore(linksToConnect[0], results);
1310             for (var i = 1; i < linksToConnect.Length; i++)
1311             {
1312                 var next = new HashSet<ulong>();
1313                 AllUsagesCore(linksToConnect[i], next);
1314                 results.IntersectWith(next);
1315             }
1316         }
1317         return results;
1318     });
1319 }
1320 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1321 {
1322     return Sync.ExecuteReadOperation(() =>
1323     {
1324         var results = new HashSet<ulong>();
1325         if (linksToConnect.Length > 0)
1326         {
1327             Links.EnsureEachLinkExists(linksToConnect);
1328             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1329             collector1.Collect(linksToConnect[0]);
1330             var next = new HashSet<ulong>();
1331             for (var i = 1; i < linksToConnect.Length; i++)
1332             {
1333                 var collector = new AllUsagesCollector(Links.Unsync, next);
1334                 collector.Collect(linksToConnect[i]);
1335                 results.IntersectWith(next);
1336                 next.Clear();
1337             }
1338         }
1339         return results;
1340     });
1341 }
1342 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1343 {
1344     return Sync.ExecuteReadOperation(() =>
1345     {
1346         var results = new HashSet<ulong>();
1347         if (linksToConnect.Length > 0)
1348         {
1349             Links.EnsureEachLinkExists(linksToConnect);
1350             var collector1 = new AllUsagesCollector(Links, results);
1351             collector1.Collect(linksToConnect[0]);
1352             //AllUsagesCore(linksToConnect[0], results);

```



```

1354         for (var i = 1; i < linksToConnect.Length; i++)
1355         {
1356             var next = new HashSet<ulong>();
1357             var collector = new AllUsagesIntersectingCollector(Links, results, next);
1358             collector.Collect(linksToConnect[i]);
1359             //AllUsagesCore(linksToConnect[i], next);
1360             //results.IntersectWith(next);
1361             results = next;
1362         }
1363     }
1364     return results;
1365 });
1366 }
1367
1368 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1369 {
1370     return Sync.ExecuteReadOperation(() =>
1371     {
1372         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1373         ↪ BitArray((int)_links.Total + 1);
1374         if (linksToConnect.Length > 0)
1375         {
1376             Links.EnsureEachLinkExists(linksToConnect);
1377             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1378             collector1.Collect(linksToConnect[0]);
1379             for (var i = 1; i < linksToConnect.Length; i++)
1380             {
1381                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1382                 ↪ BitArray((int)_links.Total + 1);
1383                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1384                 collector.Collect(linksToConnect[i]);
1385                 results = results.And(next);
1386             }
1387             return results.GetSetUInt64Indices();
1388         }
1389     });
1390 }
1391
1392 private static ulong[] Simplify(ulong[] sequence)
1393 {
1394     // Считаем новый размер последовательности
1395     long newLength = 0;
1396     var zeroOrManyStepped = false;
1397     for (var i = 0; i < sequence.Length; i++)
1398     {
1399         if (sequence[i] == ZeroOrMany)
1400         {
1401             if (zeroOrManyStepped)
1402             {
1403                 continue;
1404             }
1405             zeroOrManyStepped = true;
1406         }
1407         else
1408         {
1409             //if (zeroOrManyStepped) Is it efficient?
1410             zeroOrManyStepped = false;
1411         }
1412         newLength++;
1413     }
1414     // Строим новую последовательность
1415     zeroOrManyStepped = false;
1416     var newSequence = new ulong[newLength];
1417     long j = 0;
1418     for (var i = 0; i < sequence.Length; i++)
1419     {
1420         //var current = zeroOrManyStepped;
1421         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1422         //if (current && zeroOrManyStepped)
1423         //    continue;
1424         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1425         //if (zeroOrManyStepped && newZeroOrManyStepped)
1426         //    continue;
1427         //zeroOrManyStepped = newZeroOrManyStepped;
1428         if (sequence[i] == ZeroOrMany)
1429         {
1430             if (zeroOrManyStepped)
1431             {
1432                 continue;

```

```

1431         }
1432         zeroOrManyStepped = true;
1433     }
1434     else
1435     {
1436         //if (zeroOrManyStepped) Is it efficient?
1437         zeroOrManyStepped = false;
1438     }
1439     newSequence[j++] = sequence[i];
1440 }
1441 return newSequence;
1442 }
1443
1444 public static void TestSimplify()
1445 {
1446     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1447         ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1448     var simplifiedSequence = Simplify(sequence);
1449 }
1450
1451 public List<ulong> GetSimilarSequences() => new List<ulong>();
1452
1453 public void Prediction()
1454 {
1455     //_links
1456     //_sequences
1457 }
1458
1459 #region From Triplets
1460
1461 //public static void DeleteSequence(Link sequence)
1462 //{
1463 //}
1464
1465 public List<ulong> CollectMatchingSequences(ulong[] links)
1466 {
1467     if (links.Length == 1)
1468     {
1469         throw new Exception("Подпоследовательности с одним элементом не
1470             ↪ поддерживаются.");
1471     }
1472     var leftBound = 0;
1473     var rightBound = links.Length - 1;
1474     var left = links[leftBound++];
1475     var right = links[rightBound--];
1476     var results = new List<ulong>();
1477     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1478     return results;
1479 }
1480
1481 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1482     ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1483 {
1484     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1485     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1486     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1487     {
1488         var nextLeftLink = middleLinks[leftBound];
1489         var elements = GetRightElements(leftLink, nextLeftLink);
1490         if (leftBound <= rightBound)
1491         {
1492             for (var i = elements.Length - 1; i >= 0; i--)
1493             {
1494                 var element = elements[i];
1495                 if (element != 0)
1496                 {
1497                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1498                         ↪ rightLink, rightBound, ref results);
1499                 }
1500             }
1501         }
1502     }
1503     else
1504     {
1505         for (var i = elements.Length - 1; i >= 0; i--)
1506         {
1507             var element = elements[i];
1508             if (element != 0)
1509             {
1510                 results.Add(element);
1511             }
1512         }
1513     }
1514 }

```

```

1506     }
1507 }
1508 }
1509 }
1510 else
1511 {
1512     var nextRightLink = middleLinks[rightBound];
1513     var elements = GetLeftElements(rightLink, nextRightLink);
1514     if (leftBound <= rightBound)
1515     {
1516         for (var i = elements.Length - 1; i >= 0; i--)
1517         {
1518             var element = elements[i];
1519             if (element != 0)
1520             {
1521                 CollectMatchingSequences(leftLink, leftBound, middleLinks,
1522                     ↪ elements[i], rightBound - 1, ref results);
1523             }
1524         }
1525     }
1526     else
1527     {
1528         for (var i = elements.Length - 1; i >= 0; i--)
1529         {
1530             var element = elements[i];
1531             if (element != 0)
1532             {
1533                 results.Add(element);
1534             }
1535         }
1536     }
1537 }
1538 }
1539
1540 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1541 {
1542     var result = new ulong[5];
1543     TryStepRight(startLink, rightLink, result, 0);
1544     Links.Each(_constants.Any, startLink, couple =>
1545     {
1546         if (couple != startLink)
1547         {
1548             if (TryStepRight(couple, rightLink, result, 2))
1549             {
1550                 return false;
1551             }
1552             return true;
1553         });
1554     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1555     {
1556         result[4] = startLink;
1557     }
1558     return result;
1559 }
1560
1561 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1562 {
1563     var added = 0;
1564     Links.Each(startLink, _constants.Any, couple =>
1565     {
1566         if (couple != startLink)
1567         {
1568             var coupleTarget = Links.GetTarget(couple);
1569             if (coupleTarget == rightLink)
1570             {
1571                 result[offset] = couple;
1572                 if (++added == 2)
1573                 {
1574                     return false;
1575                 }
1576             }
1577             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1578                 ↪ == Net.And &&
1579             {
1580                 result[offset + 1] = couple;
1581                 if (++added == 2)
1582                 {

```

```

1582         return false;
1583     }
1584 }
1585 }
1586     return true;
1587 });
1588     return added > 0;
1589 }
1590
1591 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1592 {
1593     var result = new ulong[5];
1594     TryStepLeft(startLink, leftLink, result, 0);
1595     Links.Each(startLink, _constants.Any, couple =>
1596     {
1597         if (couple != startLink)
1598         {
1599             if (TryStepLeft(couple, leftLink, result, 2))
1600             {
1601                 return false;
1602             }
1603         }
1604         return true;
1605     });
1606     if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1607     {
1608         result[4] = leftLink;
1609     }
1610     return result;
1611 }
1612
1613 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1614 {
1615     var added = 0;
1616     Links.Each(_constants.Any, startLink, couple =>
1617     {
1618         if (couple != startLink)
1619         {
1620             var coupleSource = Links.GetSource(couple);
1621             if (coupleSource == leftLink)
1622             {
1623                 result[offset] = couple;
1624                 if (++added == 2)
1625                 {
1626                     return false;
1627                 }
1628             }
1629             else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1630                 ↪ == Net.And &&
1631             {
1632                 result[offset + 1] = couple;
1633                 if (++added == 2)
1634                 {
1635                     return false;
1636                 }
1637             }
1638         }
1639         return true;
1640     });
1641     return added > 0;
1642 }
1643
1644 #endregion
1645
1646 #region Walkers
1647
1648 public class PatternMatcher : RightSequenceWalker<ulong>
1649 {
1650     private readonly Sequences _sequences;
1651     private readonly ulong[] _patternSequence;
1652     private readonly HashSet<LinkIndex> _linksInSequence;
1653     private readonly HashSet<LinkIndex> _results;
1654
1655     #region Pattern Match
1656
1657     enum PatternBlockType
1658     {
1659         Undefined,
1660         Gap,
1661         Elements
1662     }

```

```

1661 }
1662
1663 struct PatternBlock
1664 {
1665     public PatternBlockType Type;
1666     public long Start;
1667     public long Stop;
1668 }
1669
1670 private readonly List<PatternBlock> _pattern;
1671 private int _patternPosition;
1672 private long _sequencePosition;
1673
1674 #endregion
1675
1676 public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1677     ↳ HashSet<LinkIndex> results)
1678     : base(sequences.Links.Unsync)
1679 {
1680     _sequences = sequences;
1681     _patternSequence = patternSequence;
1682     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1683     ↳ _constants.Any && x != ZeroOrMany));
1684     _results = results;
1685     _pattern = CreateDetailedPattern();
1686 }
1687
1688 protected override bool IsElement(ICollection<ulong> link) =>
1689     ↳ _linksInSequence.Contains(Links.GetIndex(link)) || base.IsElement(link);
1690
1691 public bool PatternMatch(LinkIndex sequenceToMatch)
1692 {
1693     _patternPosition = 0;
1694     _sequencePosition = 0;
1695     foreach (var part in Walk(sequenceToMatch))
1696     {
1697         if (!PatternMatchCore(Links.GetIndex(part)))
1698         {
1699             break;
1700         }
1701     }
1702     return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1703     ↳ - 1 && _pattern[_patternPosition].Start == 0);
1704 }
1705
1706 private List<PatternBlock> CreateDetailedPattern()
1707 {
1708     var pattern = new List<PatternBlock>();
1709     var patternBlock = new PatternBlock();
1710     for (var i = 0; i < _patternSequence.Length; i++)
1711     {
1712         if (patternBlock.Type == PatternBlockType.Undefined)
1713         {
1714             if (_patternSequence[i] == _constants.Any)
1715             {
1716                 patternBlock.Type = PatternBlockType.Gap;
1717                 patternBlock.Start = 1;
1718                 patternBlock.Stop = 1;
1719             }
1720             else if (_patternSequence[i] == ZeroOrMany)
1721             {
1722                 patternBlock.Type = PatternBlockType.Gap;
1723                 patternBlock.Start = 0;
1724                 patternBlock.Stop = long.MaxValue;
1725             }
1726             else
1727             {
1728                 patternBlock.Type = PatternBlockType.Elements;
1729                 patternBlock.Start = i;
1730                 patternBlock.Stop = i;
1731             }
1732         }
1733         else if (patternBlock.Type == PatternBlockType.Elements)
1734         {
1735             if (_patternSequence[i] == _constants.Any)
1736             {
1737                 pattern.Add(patternBlock);
1738                 patternBlock = new PatternBlock
1739                 {
1740                     Type = PatternBlockType.Gap,

```

```

1737         Start = 1,
1738         Stop = 1
1739     };
1740 }
1741 else if (_patternSequence[i] == ZeroOrMany)
1742 {
1743     pattern.Add(patternBlock);
1744     patternBlock = new PatternBlock
1745     {
1746         Type = PatternBlockType.Gap,
1747         Start = 0,
1748         Stop = long.MaxValue
1749     };
1750 }
1751 else
1752 {
1753     patternBlock.Stop = i;
1754 }
1755 }
1756 else // patternBlock.Type == PatternBlockType.Gap
1757 {
1758     if (_patternSequence[i] == _constants.Any)
1759     {
1760         patternBlock.Start++;
1761         if (patternBlock.Stop < patternBlock.Start)
1762         {
1763             patternBlock.Stop = patternBlock.Start;
1764         }
1765     }
1766     else if (_patternSequence[i] == ZeroOrMany)
1767     {
1768         patternBlock.Stop = long.MaxValue;
1769     }
1770     else
1771     {
1772         pattern.Add(patternBlock);
1773         patternBlock = new PatternBlock
1774         {
1775             Type = PatternBlockType.Elements,
1776             Start = i,
1777             Stop = i
1778         };
1779     }
1780 }
1781 }
1782 if (patternBlock.Type != PatternBlockType.Undefined)
1783 {
1784     pattern.Add(patternBlock);
1785 }
1786 return pattern;
1787 }
1788
1789 /* match: search for regexp anywhere in text */
1790 int match(char* regexp, char* text)
1791 {
1792     do
1793     {
1794         } while (*text++ != '\0');
1795     return 0;
1796 }
1797
1798 /* matchhere: search for regexp at beginning of text */
1799 int matchhere(char* regexp, char* text)
1800 {
1801     if (regexp[0] == '\0')
1802         return 1;
1803     if (regexp[1] == '*')
1804         return matchstar(regexp[0], regexp + 2, text);
1805     if (regexp[0] == '$' && regexp[1] == '\0')
1806         return *text == '\0';
1807     if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1808         return matchhere(regexp + 1, text + 1);
1809     return 0;
1810 }
1811
1812 /* matchstar: search for c*regexp at beginning of text */
1813 int matchstar(int c, char* regexp, char* text)
1814 {
1815     do

```

```

1816 // { /* a * matches zero or more instances */
1817 //     if (matchhere(regexp, text))
1818 //         return 1;
1819 // } while (*text != '\0' && (*text++ == c || c == '.'));
1820 // return 0;
1821 //}
1822
1823 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1824 //    ↳ long maximumGap)
1825 //{
1826 //    mininumGap = 0;
1827 //    maximumGap = 0;
1828 //    element = 0;
1829 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1830 //    {
1831 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1832 //            mininumGap++;
1833 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1834 //            maximumGap = long.MaxValue;
1835 //        else
1836 //            break;
1837 //    }
1838 //    if (maximumGap < mininumGap)
1839 //        maximumGap = mininumGap;
1840 //}
1841
1842 private bool PatternMatchCore(LinkIndex element)
1843 {
1844     if (_patternPosition >= _pattern.Count)
1845     {
1846         _patternPosition = -2;
1847         return false;
1848     }
1849     var currentPatternBlock = _pattern[_patternPosition];
1850     if (currentPatternBlock.Type == PatternBlockType.Gap)
1851     {
1852         //var currentMatchingBlockLength = (_sequencePosition -
1853         //    ↳ _lastMatchedBlockPosition);
1854         if (_sequencePosition < currentPatternBlock.Start)
1855         {
1856             _sequencePosition++;
1857             return true; // Двигаемся дальше
1858         }
1859         // Это последний блок
1860         if (_pattern.Count == _patternPosition + 1)
1861         {
1862             _patternPosition++;
1863             _sequencePosition = 0;
1864             return false; // Полное соответствие
1865         }
1866         else
1867         {
1868             if (_sequencePosition > currentPatternBlock.Stop)
1869             {
1870                 return false; // Соответствие невозможно
1871             }
1872             var nextPatternBlock = _pattern[_patternPosition + 1];
1873             if (_patternSequence[nextPatternBlock.Start] == element)
1874             {
1875                 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1876                 {
1877                     _patternPosition++;
1878                     _sequencePosition = 1;
1879                 }
1880                 else
1881                 {
1882                     _patternPosition += 2;
1883                     _sequencePosition = 0;
1884                 }
1885             }
1886         }
1887     }
1888     else // currentPatternBlock.Type == PatternBlockType.Elements
1889     {
1890         var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1891         if (_patternSequence[patternElementPosition] != element)
1892         {
1893             return false; // Соответствие невозможно

```

```

1893     }
1894     if (patternElementPosition == currentPatternBlock.Stop)
1895     {
1896         _patternPosition++;
1897         _sequencePosition = 0;
1898     }
1899     else
1900     {
1901         _sequencePosition++;
1902     }
1903 }
1904 return true;
1905 //if (_patternSequence[_patternPosition] != element)
1906 //    return false;
1907 //else
1908 //{
1909 //    _sequencePosition++;
1910 //    _patternPosition++;
1911 //    return true;
1912 //}
1913 ///////
1914 //if (_filterPosition == _patternSequence.Length)
1915 //{
1916 //    _filterPosition = -2; // Длиннее чем нужно
1917 //    return false;
1918 //}
1919 //if (element != _patternSequence[_filterPosition])
1920 //{
1921 //    _filterPosition = -1;
1922 //    return false; // Начинается иначе
1923 //}
1924 //_filterPosition++;
1925 //if (_filterPosition == (_patternSequence.Length - 1))
1926 //    return false;
1927 //if (_filterPosition >= 0)
1928 //{
1929 //    if (element == _patternSequence[_filterPosition + 1])
1930 //        _filterPosition++;
1931 //    else
1932 //        return false;
1933 //}
1934 //if (_filterPosition < 0)
1935 //{
1936 //    if (element == _patternSequence[0])
1937 //        _filterPosition = 0;
1938 //}
1939 }
1940
1941 public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
1942 {
1943     foreach (var sequenceToMatch in sequencesToMatch)
1944     {
1945         if (PatternMatch(sequenceToMatch))
1946         {
1947             _results.Add(sequenceToMatch);
1948         }
1949     }
1950 }
1951 }
1952
1953 #endregion
1954 }
1955 }

```

./Sequences/Sequences.Experiments.ReadSequence.cs

```

1  // #define USEARRAYPOOL
2  using System;
3  using System.Runtime.CompilerServices;
4  #if USEARRAYPOOL
5  using Platform.Collections;
6  #endif
7
8  namespace Platform.Data.Doublets.Sequences
9  {
10     partial class Sequences
11     {
12         public ulong[] ReadSequenceCore(ulong sequence, Func<ulong, bool> isElement)
13         {
14             var links = Links.Unsync;

```



```

15     var length = 1;
16     var array = new ulong[length];
17     array[0] = sequence;
18
19     if (isElement(sequence))
20     {
21         return array;
22     }
23
24     bool hasElements;
25     do
26     {
27         length *= 2;
28 #if USEARRAYPOOL
29         var nextArray = ArrayPool.Allocate<ulong>(length);
30 #else
31         var nextArray = new ulong[length];
32 #endif
33         hasElements = false;
34         for (var i = 0; i < array.Length; i++)
35         {
36             var candidate = array[i];
37             if (candidate == 0)
38             {
39                 continue;
40             }
41             var doubletOffset = i * 2;
42             if (isElement(candidate))
43             {
44                 nextArray[doubletOffset] = candidate;
45             }
46             else
47             {
48                 var link = links.GetLink(candidate);
49                 var linkSource = links.GetSource(link);
50                 var linkTarget = links.GetTarget(link);
51                 nextArray[doubletOffset] = linkSource;
52                 nextArray[doubletOffset + 1] = linkTarget;
53                 if (!hasElements)
54                 {
55                     hasElements = !(isElement(linkSource) && isElement(linkTarget));
56                 }
57             }
58         }
59 #if USEARRAYPOOL
60         if (array.Length > 1)
61         {
62             ArrayPool.Free(array);
63         }
64 #endif
65         array = nextArray;
66     }
67     while (hasElements);
68     var filledElementsCount = CountFilledElements(array);
69     if (filledElementsCount == array.Length)
70     {
71         return array;
72     }
73     else
74     {
75         return CopyFilledElements(array, filledElementsCount);
76     }
77 }
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 private static ulong[] CopyFilledElements(ulong[] array, int filledElementsCount)
81 {
82     var finalArray = new ulong[filledElementsCount];
83     for (int i = 0, j = 0; i < array.Length; i++)
84     {
85         if (array[i] > 0)
86         {
87             finalArray[j] = array[i];
88             j++;
89         }
90     }
91 #if USEARRAYPOOL
92     ArrayPool.Free(array);
93 #endif
94     return finalArray;

```

```

95     }
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     private static int CountFilledElements(ulong[] array)
99     {
100         var count = 0;
101         for (var i = 0; i < array.Length; i++)
102         {
103             if (array[i] > 0)
104             {
105                 count++;
106             }
107         }
108         return count;
109     }
110 }
111 }

```

./Sequences/SequencesExtensions.cs

```

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

```

./Sequences/SequencesIndexer.cs

```

1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Sequences
4  {
5      public class SequencesIndexer<TLink>
6      {
7          private static readonly EqualityComparer<TLink> _equalityComparer =
8              ↳ EqualityComparer<TLink>.Default;
9
10         private readonly ISynchronizedLinks<TLink> _links;
11         private readonly TLink _null;
12
13         public SequencesIndexer(ISynchronizedLinks<TLink> links)
14         {
15             _links = links;
16             _null = _links.Constants.Null;
17         }
18
19         /// <summary>
20         /// Индексирует последовательность глобально, и возвращает значение,
21         /// определяющие была ли запрошенная последовательность проиндексирована ранее.
22         /// </summary>
23         /// <param name="sequence">Последовательность для индексации.</param>
24         /// <returns>
25         /// True если последовательность уже была проиндексирована ранее и
26         /// False если последовательность была проиндексирована только что.
27         /// </returns>
28         public bool Index(TLink[] sequence)
29         {
30             var indexed = true;
31             var i = sequence.Length;
32             while (--i >= 1 && (indexed =
33                 ↳ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
34                 ↳ _null))) { }
35             for (; i >= 1; i--)
36             {
37                 _links.GetOrCreate(sequence[i - 1], sequence[i]);
38             }
39         }

```

```

36         return indexed;
37     }
38
39     public bool BulkIndex(TLink[] sequence)
40     {
41         var indexed = true;
42         var i = sequence.Length;
43         var links = _links.Unsync;
44         _links.SyncRoot.ExecuteReadOperation(() =>
45         {
46             while (--i >= 1 && (indexed =
47                 ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
48                 ↪ sequence[i]), _null))) { }
49         });
50         if (indexed == false)
51         {
52             _links.SyncRoot.ExecuteWriteOperation(() =>
53             {
54                 for (; i >= 1; i--)
55                 {
56                     links.GetOrCreate(sequence[i - 1], sequence[i]);
57                 }
58             });
59         }
60         return indexed;
61     }
62
63     public bool BulkIndexUnsync(TLink[] sequence)
64     {
65         var indexed = true;
66         var i = sequence.Length;
67         var links = _links.Unsync;
68         while (--i >= 1 && (indexed =
69             ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1], sequence[i]),
70             ↪ _null))) { }
71         for (; i >= 1; i--)
72         {
73             links.GetOrCreate(sequence[i - 1], sequence[i]);
74         }
75         return indexed;
76     }
77
78     public bool CheckIndex(IList<TLink> sequence)
79     {
80         var indexed = true;
81         var i = sequence.Count;
82         while (--i >= 1 && (indexed =
83             ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
84             ↪ _null))) { }
85         return indexed;
86     }
87 }

```

./Sequences/SequencesOptions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
5  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
6  using Platform.Data.Doublets.Sequences.Converters;
7  using Platform.Data.Doublets.Sequences.CriteriaMatchers;
8
9  namespace Platform.Data.Doublets.Sequences
10 {
11     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
12     ↪ ILinks<TLink> must contain GetConstants function.
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15         ↪ EqualityComparer<TLink>.Default;
16
17         public TLink SequenceMarkerLink { get; set; }
18         public bool UseCascadeUpdate { get; set; }
19         public bool UseCascadeDelete { get; set; }
20         public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
21         public bool UseSequenceMarker { get; set; }
22         public bool UseCompression { get; set; }
23         public bool UseGarbageCollection { get; set; }
24         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }

```

```

23 public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
24
25 public MarkedSequenceCriteriaMatcher<TLink> MarkedSequenceMatcher { get; set; }
26 public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
27 public SequencesIndexer<TLink> Indexer { get; set; }
28
29 // TODO: Реализовать компактификацию при чтении
30 //public bool EnforceSingleSequenceVersionOnRead { get; set; }
31 //public bool UseRequestMarker { get; set; }
32 //public bool StoreRequestResults { get; set; }
33
34 public void InitOptions(ISynchronizedLinks<TLink> links)
35 {
36     if (UseSequenceMarker)
37     {
38         if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
39         {
40             SequenceMarkerLink = links.CreatePoint();
41         }
42         else
43         {
44             if (!links.Exists(SequenceMarkerLink))
45             {
46                 var link = links.CreatePoint();
47                 if (!_equalityComparer.Equals(link, SequenceMarkerLink))
48                 {
49                     throw new InvalidOperationException("Cannot recreate sequence marker
50                     ↪ link.");
51                 }
52             }
53             if (MarkedSequenceMatcher == null)
54             {
55                 MarkedSequenceMatcher = new MarkedSequenceCriteriaMatcher<TLink>(links,
56                 ↪ SequenceMarkerLink);
57             }
58             var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
59             if (UseCompression)
60             {
61                 if (LinksToSequenceConverter == null)
62                 {
63                     ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
64                     if (UseSequenceMarker)
65                     {
66                         totalSequenceSymbolFrequencyCounter = new
67                         ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
68                         ↪ MarkedSequenceMatcher);
69                     }
70                     else
71                     {
72                         totalSequenceSymbolFrequencyCounter = new
73                         ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
74                     }
75                     var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
76                     ↪ totalSequenceSymbolFrequencyCounter);
77                     var compressingConverter = new CompressingConverter<TLink>(links,
78                     ↪ balancedVariantConverter, doubletFrequenciesCache);
79                     LinksToSequenceConverter = compressingConverter;
80                 }
81             }
82             else
83             {
84                 if (LinksToSequenceConverter == null)
85                 {
86                     LinksToSequenceConverter = balancedVariantConverter;
87                 }
88             }
89             if (UseIndex && Indexer == null)
90             {
91                 Indexer = new SequencesIndexer<TLink>(links);
92             }
93         }
94     }
95
96     public void ValidateOptions()
97     {
98         if (UseGarbageCollection && !UseSequenceMarker)
99         {
100

```

```

94         throw new NotSupportedException("To use garbage collection UseSequenceMarker
           ↪ option must be on.");
95     }
96 }
97 }
98 }

```

./Sequences/UnicodeMap.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using Platform.Data.Sequences;
7
8  namespace Platform.Data.Doublets.Sequences
9  {
10     public class UnicodeMap
11     {
12         public static readonly ulong FirstCharLink = 1;
13         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
14         public static readonly ulong MapSize = 1 + char.MaxValue;
15
16         private readonly ILinks<ulong> _links;
17         private bool _initialized;
18
19         public UnicodeMap(ILinks<ulong> links) => _links = links;
20
21         public static UnicodeMap InitNew(ILinks<ulong> links)
22         {
23             var map = new UnicodeMap(links);
24             map.Init();
25             return map;
26         }
27
28         public void Init()
29         {
30             if (_initialized)
31             {
32                 return;
33             }
34             _initialized = true;
35             var firstLink = _links.CreatePoint();
36             if (firstLink != FirstCharLink)
37             {
38                 _links.Delete(firstLink);
39             }
40             else
41             {
42                 for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
43                 {
44                     // From NIL to It (NIL -> Character) transformation meaning, (or infinite
45                     ↪ amount of NIL characters before actual Character)
46                     var createdLink = _links.CreatePoint();
47                     _links.Update(createdLink, firstLink, createdLink);
48                     if (createdLink != i)
49                     {
50                         throw new InvalidOperationException("Unable to initialize UTF 16
51                         ↪ table.");
52                     }
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) => (ulong)character + 1;
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         public static char FromLinkToChar(ulong link) => (char)(link - 1);
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         public static bool IsCharLink(ulong link) => link <= MapSize;
70
71         public static string FromLinksToString(IList<ulong> linksList)
72         {
73

```

```

71     var sb = new StringBuilder();
72     for (int i = 0; i < linksList.Count; i++)
73     {
74         sb.Append(FromLinkToChar(linksList[i]));
75     }
76     return sb.ToString();
77 }
78
79 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
80 {
81     var sb = new StringBuilder();
82     if (links.Exists(link))
83     {
84         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
85             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
86             element =>
87             {
88                 sb.Append(FromLinkToChar(element));
89                 return true;
90             }
91         );
92     }
93     return sb.ToString();
94 }
95
96 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
97     ↪ chars.Length);
98
99 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
100 {
101     // char array to ulong array
102     var linksSequence = new ulong[count];
103     for (var i = 0; i < count; i++)
104     {
105         linksSequence[i] = FromCharToLink(chars[i]);
106     }
107     return linksSequence;
108 }
109
110 public static ulong[] FromStringToLinkArray(string sequence)
111 {
112     // char array to ulong array
113     var linksSequence = new ulong[sequence.Length];
114     for (var i = 0; i < sequence.Length; i++)
115     {
116         linksSequence[i] = FromCharToLink(sequence[i]);
117     }
118     return linksSequence;
119 }
120
121 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
122 {
123     var result = new List<ulong[]>();
124     var offset = 0;
125     while (offset < sequence.Length)
126     {
127         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
128         var relativeLength = 1;
129         var absoluteLength = offset + relativeLength;
130         while (absoluteLength < sequence.Length &&
131             currentCategory ==
132             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
133         {
134             relativeLength++;
135             absoluteLength++;
136         }
137         // char array to ulong array
138         var innerSequence = new ulong[relativeLength];
139         var maxLength = offset + relativeLength;
140         for (var i = offset; i < maxLength; i++)
141         {
142             innerSequence[i - offset] = FromCharToLink(sequence[i]);
143         }
144         result.Add(innerSequence);
145         offset += relativeLength;
146     }
147     return result;
148 }
149
150 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)

```

```

147 {
148     var result = new List<ulong[]>();
149     var offset = 0;
150     while (offset < array.Length)
151     {
152         var relativeLength = 1;
153         if (array[offset] <= LastCharLink)
154         {
155             var currentCategory =
156                 ↳ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
157             var absoluteLength = offset + relativeLength;
158             while (absoluteLength < array.Length &&
159                 array[absoluteLength] <= LastCharLink &&
160                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
161                 ↳ array[absoluteLength])))
162             {
163                 relativeLength++;
164                 absoluteLength++;
165             }
166         }
167         else
168         {
169             var absoluteLength = offset + relativeLength;
170             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
171             {
172                 relativeLength++;
173                 absoluteLength++;
174             }
175             // copy array
176             var innerSequence = new ulong[relativeLength];
177             var maxLength = offset + relativeLength;
178             for (var i = offset; i < maxLength; i++)
179             {
180                 innerSequence[i - offset] = array[i];
181             }
182             result.Add(innerSequence);
183             offset += relativeLength;
184         }
185     }
186     return result;
187 }

```

./Sequences/Walkers/LeftSequenceWalker.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Data.Doublets.Sequences.Walkers
5  {
6      public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
7      {
8          public LeftSequenceWalker(ILinks<TLink> links) : base(links) { }
9
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         protected override IList<TLink> GetNextElementAfterPop(IList<TLink> element) =>
12             ↳ Links.GetLink(Links.GetSource(element));
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override IList<TLink> GetNextElementAfterPush(IList<TLink> element) =>
16             ↳ Links.GetLink(Links.GetTarget(element));
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override IEnumerable<IList<TLink>> WalkContents(IList<TLink> element)
20         {
21             var start = Links.Constants.IndexPart + 1;
22             for (var i = element.Count - 1; i >= start; i--)
23             {
24                 var partLink = Links.GetLink(element[i]);
25                 if (IsElement(partLink))
26                 {
27                     yield return partLink;
28                 }
29             }
30         }
31     }
32 }

```

./Sequences/Walkers/RightSequenceWalker.cs

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 namespace Platform.Data.Doublets.Sequences.Walkers
5 {
6     public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
7     {
8         public RightSequenceWalker(ILinks<TLink> links) : base(links) { }
9
10        [MethodImpl(MethodImplOptions.AggressiveInlining)]
11        protected override IList<TLink> GetNextElementAfterPop(IList<TLink> element) =>
12            ↪ Links.GetLink(Links.GetTarget(element));
13
14        [MethodImpl(MethodImplOptions.AggressiveInlining)]
15        protected override IList<TLink> GetNextElementAfterPush(IList<TLink> element) =>
16            ↪ Links.GetLink(Links.GetSource(element));
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        protected override IEnumerable<IList<TLink>> WalkContents(IList<TLink> element)
20        {
21            for (var i = Links.Constants.IndexPart + 1; i < element.Count; i++)
22            {
23                var partLink = Links.GetLink(element[i]);
24                if (IsElement(partLink))
25                {
26                    yield return partLink;
27                }
28            }
29        }
30    }
31 }
```

./Sequences/Walkers/SequenceWalkerBase.cs

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



```

43         {
44             _stack.Push(element);
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> GetNextElementAfterPop(IList<TLink> element);
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 protected abstract IList<TLink> GetNextElementAfterPush(IList<TLink> element);
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected abstract IEnumerable<IList<TLink>> WalkContents(IList<TLink> element);
63 }

```

./Stacks/Stack.cs

```

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

```

./Stacks/StackExtensions.cs

```

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

```

```

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

```

./SynchronizedLinks.cs

```

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

```

./UInt64Link.cs

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Platform.Exceptions;
5  using Platform.Ranges;
6  using Platform.Helpers.Singletons;
7  using Platform.Data.Constants;
8
9  namespace Platform.Data.Doublets
10 {
11     /// <summary>
12     /// Структура описывающая уникальную связь.
13     /// </summary>
14     public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
15     {
16         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
17             ↪ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;

```

```

17
18 private const int Length = 3;
19
20 public readonly ulong Index;
21 public readonly ulong Source;
22 public readonly ulong Target;
23
24 public static readonly UInt64Link Null = new UInt64Link();
25
26 public UInt64Link(params ulong[] values)
27 {
28     Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
29         ↪ _constants.Null;
30     Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
31         ↪ _constants.Null;
32     Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
33         ↪ _constants.Null;
34 }
35
36 public UInt64Link(IList<ulong> values)
37 {
38     Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
39         ↪ _constants.Null;
40     Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
41         ↪ _constants.Null;
42     Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
43         ↪ _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) => new UInt64Link(source,
61     ↪ target);
62
63 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
64
65 public bool IsNull() => Index == _constants.Null
66     && Source == _constants.Null
67     && Target == _constants.Null;
68
69 public override bool Equals(object other) => other is UInt64Link &&
70     ↪ Equals((UInt64Link)other);
71
72 public bool Equals(UInt64Link other) => Index == other.Index
73     && Source == other.Source
74     && Target == other.Target;
75
76 public static string ToString(ulong index, ulong source, ulong target) => $"({index}:
77     ↪ {source}->{target})";
78
79 public static string ToString(ulong source, ulong target) => $"({source}->{target})";
80
81 public static implicit operator ulong[] (UInt64Link link) => link.ToArray();
82
83 public static implicit operator UInt64Link(ulong[] linkArray) => new
84     ↪ UInt64Link(linkArray);
85
86 public ulong[] ToArray()
87 {
88     var array = new ulong[Length];
89     CopyTo(array, 0);
90     return array;
91 }
92
93 public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
94     ↪ : ToString(Index, Source, Target);
95
96 #region IList

```

```

86
87 public ulong this[int index]
88 {
89     get
90     {
91         Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
92             ↳ nameof(index));
93         if (index == _constants.IndexPart)
94         {
95             return Index;
96         }
97         if (index == _constants.SourcePart)
98         {
99             return Source;
100         }
101         if (index == _constants.TargetPart)
102         {
103             return Target;
104         }
105         throw new NotSupportedException(); // Impossible path due to
106             ↳ Ensure.ArgumentInRange
107     }
108     set => throw new NotSupportedException();
109 }
110
111 public int Count => Length;
112
113 public bool IsReadOnly => true;
114
115 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
116
117 public IEnumerator<ulong> GetEnumerator()
118 {
119     yield return Index;
120     yield return Source;
121     yield return Target;
122 }
123
124 public void Add(ulong item) => throw new NotSupportedException();
125
126 public void Clear() => throw new NotSupportedException();
127
128 public bool Contains(ulong item) => IndexOf(item) >= 0;
129
130 public void CopyTo(ulong[] array, int arrayIndex)
131 {
132     Ensure.Always.ArgumentNotNull(array, nameof(array));
133     Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
134         ↳ nameof(arrayIndex));
135     if (arrayIndex + Length > array.Length)
136     {
137         throw new ArgumentException();
138     }
139     array[arrayIndex++] = Index;
140     array[arrayIndex++] = Source;
141     array[arrayIndex] = Target;
142 }
143
144 public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
145
146 public int IndexOf(ulong item)
147 {
148     if (Index == item)
149     {
150         return _constants.IndexPart;
151     }
152     if (Source == item)
153     {
154         return _constants.SourcePart;
155     }
156     if (Target == item)
157     {
158         return _constants.TargetPart;
159     }
160     return -1;
161 }
162
163 public void Insert(int index, ulong item) => throw new NotSupportedException();

```

```

163         public void RemoveAt(int index) => throw new NotSupportedException();
164     }
165     #endregion
166 }
167 }

```

./UInt64LinkExtensions.cs

```

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

```

./UInt64LinksExtensions.cs

```

1 using System;
2 using System.Text;
3 using System.Collections.Generic;
4 using Platform.Helpers.Singletons;
5 using Platform.Data.Constants;
6 using Platform.Data.Exceptions;
7 using Platform.Data.Doublets.Sequences;
8
9 namespace Platform.Data.Doublets
10 {
11     public static class UInt64LinksExtensions
12     {
13         public static readonly LinksCombinedConstants<bool, ulong, int> Constants =
14             ⇨ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
15
16         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
17
18         public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
19         {
20             if (sequence == null)
21             {
22                 return;
23             }
24             for (var i = 0; i < sequence.Count; i++)
25             {
26                 if (!links.Exists(sequence[i]))
27                 {
28                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
29                         ⇨ $"sequence[{i}]");
30                 }
31             }
32         }
33
34         public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
35             ⇨ sequence)
36         {
37             if (sequence == null)
38             {
39                 return;
40             }
41             for (var i = 0; i < sequence.Count; i++)
42             {
43                 if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
44                 {
45                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
46                         ⇨ $"sequence[{i}]");
47                 }
48             }
49         }
50
51         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
52         {
53             if (sequence == null)
54             {
55                 return false;
56             }
57             var constants = links.Constants;
58             for (var i = 0; i < sequence.Length; i++)
59             {
60                 if (sequence[i] == constants.Any)
61                 {
62                     return true;
63                 }
64             }
65             return false;
66         }
67     }
68 }

```

```

58         return true;
59     }
60 }
61 return false;
62 }
63
64 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
    ↪ Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
65 {
66     var sb = new StringBuilder();
67     var visited = new HashSet<ulong>();
68     links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
    ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
69     return sb.ToString();
70 }
71
72 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
    ↪ Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
    ↪ bool renderIndex = false, bool renderDebug = false)
73 {
74     var sb = new StringBuilder();
75     var visited = new HashSet<ulong>();
76     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
    ↪ renderDebug);
77     return sb.ToString();
78 }
79
80 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
    ↪ HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
    ↪ Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false, bool
    ↪ renderDebug = false)
81 {
82     if (sb == null)
83     {
84         throw new ArgumentNullException(nameof(sb));
85     }
86     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
    ↪ Constants.Itself)
87     {
88         return;
89     }
90     if (links.Exists(linkIndex))
91     {
92         if (visited.Add(linkIndex))
93         {
94             sb.Append('(');
95             var link = new UInt64Link(links.GetLink(linkIndex));
96             if (renderIndex)
97             {
98                 sb.Append(link.Index);
99                 sb.Append(':');
100             }
101             if (link.Source == link.Index)
102             {
103                 sb.Append(link.Index);
104             }
105             else
106             {
107                 var source = new UInt64Link(links.GetLink(link.Source));
108                 if (isElement(source))
109                 {
110                     appendElement(sb, source);
111                 }
112                 else
113                 {
114                     links.AppendStructure(sb, visited, source.Index, isElement,
    ↪ appendElement, renderIndex);
115                 }
116             }
117             sb.Append(' ');
118             if (link.Target == link.Index)
119             {
120                 sb.Append(link.Index);
121             }
122             else
123             {
124                 var target = new UInt64Link(links.GetLink(link.Target));
125                 if (isElement(target))

```

```

126         {
127             appendElement(sb, target);
128         }
129         else
130         {
131             links.AppendStructure(sb, visited, target.Index, isElement,
132                 ↪ appendElement, renderIndex);
133         }
134         sb.Append(')');
135     }
136     else
137     {
138         if (renderDebug)
139         {
140             sb.Append('*');
141         }
142         sb.Append(linkIndex);
143     }
144 }
145 else
146 {
147     if (renderDebug)
148     {
149         sb.Append('~');
150     }
151     sb.Append(linkIndex);
152 }
153 }
154 }
155 }

```

./UInt64LinksTransactionsLayer.cs

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Runtime.CompilerServices;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Platform.Disposables;
9  using Platform.Timestamps;
10 using Platform.Unsafe;
11 using Platform.IO;
12 using Platform.Data.Doublets.Decorators;
13
14 namespace Platform.Data.Doublets
15 {
16     public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase

```

```

47     /// {
48     ///     get
49     ///     {
50     ///         return (ulong) mask & TransactionIdCombined;
51     ///     }
52     /// }
53     ///
54     /// public UniqueTimestamp Timestamp
55     /// {
56     ///     get
57     ///     {
58     ///         return (UniqueTimestamp)mask & TransactionIdCombined;
59     ///     }
60     /// }
61     ///
62     /// public TransactionItemType Type
63     /// {
64     ///     get
65     ///     {
66     ///         // Использовать по одному биту из TransactionId и Timestamp,
67     ///         // для значения в 2 бита, которое представляет тип операции
68     ///         throw new NotImplementedException();
69     ///     }
70     /// }
71     /// }
72     ///
73     /// private struct Transition
74     /// {
75     ///     public TransitionHeader Header;
76     ///     public Link Source;
77     ///     public Link Linker;
78     ///     public Link Target;
79     /// }
80     ///
81     /// </remarks>
82     public struct Transition
83     {
84         public static readonly long Size = StructureHelpers.SizeOf<Transition>();
85
86         public readonly ulong TransactionId;
87         public readonly UInt64Link Before;
88         public readonly UInt64Link After;
89         public readonly Timestamp Timestamp;
90
91         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
92             ↪ transactionId, UInt64Link before, UInt64Link after)
93         {
94             TransactionId = transactionId;
95             Before = before;
96             After = after;
97             Timestamp = uniqueTimestampFactory.Create();
98         }
99
100         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
101             ↪ transactionId, UInt64Link before)
102             : this(uniqueTimestampFactory, transactionId, before, default)
103         {
104         }
105
106         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
107             : this(uniqueTimestampFactory, transactionId, default, default)
108         {
109         }
110
111         public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
112             ↪ {After}";
113     }
114
115     /// <remarks>
116     /// Другие варианты реализации транзакций (атомарности):
117     /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
118     ///    ↪ Target)) и индексов.
119     /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
120     ///    ↪ потребуется решить вопрос
121     ///    со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
122     ///    ↪ пересечениями идентификаторов.
123     ///
124     /// Где хранить промежуточный список транзакций?

```



```

119 ///
120 /// В оперативной памяти:
121 /// Минусы:
122 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
123 /// так как нужно отдельно выделять память под список трансформаций.
124 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
125 /// если транзакция использует слишком много трансформаций.
126 /// -> Можно использовать жёсткий диск для слишком длинных транзакций.
127 /// -> Максимальный размер списка трансформаций можно ограничить / задать
    ↪ константой.
128 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
    ↪ создавая задержку.
129 ///
130 /// На жёстком диске:
131 /// Минусы:
132 /// 1. Длительный отклик, на запись каждой трансформации.
133 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
134 /// -> Это может решаться упаковкой/исключением дублирующих операций.
135 /// -> Также это может решаться тем, что короткие транзакции вообще
136 /// не будут записываться в случае отката.
137 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
    ↪ операции (трансформации)
    ↪ будут записаны в лог.
138 ///
139 ///
140 /// </remarks>
141 public class Transaction : DisposableBase
142 {
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 transactions not supported.");
154         }
155         IsCommitted = false;
156         IsReverted = false;
157         _transitions = new Queue<Transition>();
158         SetCurrentTransaction(layer, this);
159     }
160
161     public void Commit()
162     {
163         EnsureTransactionAllowsWriteOperations(this);
164         while (_transitions.Count > 0)
165         {
166             var transition = _transitions.Dequeue();
167             _layer._transitions.Enqueue(transition);
168         }
169         _layer._lastCommittedTransactionId = _layer._currentTransactionId;
170         IsCommitted = true;
171     }
172
173     private void Revert()
174     {
175         EnsureTransactionAllowsWriteOperations(this);
176         var transitionsToRevert = new Transition[_transitions.Count];
177         _transitions.CopyTo(transitionsToRevert, 0);
178         for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
179         {
180             _layer.RevertTransition(transitionsToRevert[i]);
181         }
182         IsReverted = true;
183     }
184
185     public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
        ↪ Transaction transaction)
186     {
187         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
188         layer._currentTransactionTransitions = transaction._transitions;
189         layer._currentTransaction = transaction;
190     }
191
192     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
193     {

```

```

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

```

```

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(_currentTransaction);
301     }
302     var transitions = GetCurrentTransitions();
303     transitions.Enqueue(transition);
304 }
305
306 private void RevertTransition(Transition transition)
307 {
308     if (transition.After.IsNull()) // Revert Deletion with Creation
309     {
310         Links.Create();
311     }
312     else if (transition.Before.IsNull()) // Revert Creation with Deletion
313     {
314         Links.Delete(transition.After.Index);
315     }
316     else // Revert Update
317     {
318         Links.Update(new[] { transition.After.Index, transition.Before.Source,
    ↪     transition.Before.Target });
319     }
320 }
321
322 private void ResetCurrentTransation()
323 {
324     _currentTransactionId = 0;
325     _currentTransactionTransitions = null;
326     _currentTransaction = null;
327 }
328
329 private void PushTransitions()
330 {
331     if (_log == null || _transitions == null)
332     {
333         return;
334     }
335     for (var i = 0; i < _transitions.Count; i++)
336     {
337         var transition = _transitions.Dequeue();
338
339         _log.Write(transition);
340         _lastCommittedTransition = transition;
341     }
342 }
343

```

```

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

```

Index

- ./Converters/AddressToUnaryNumberConverter.cs, 1
- ./Converters/LinkToItsFrequencyNumberConverter.cs, 1
- ./Converters/PowerOf2ToUnaryNumberConverter.cs, 2
- ./Converters/UnaryNumberToAddressAddOperationConverter.cs, 2
- ./Converters/UnaryNumberToAddressOrOperationConverter.cs, 3
- ./Decorators/LinksCascadeDependenciesResolver.cs, 4
- ./Decorators/LinksCascadeUniquenessAndDependenciesResolver.cs, 4
- ./Decorators/LinksDecoratorBase.cs, 5
- ./Decorators/LinksDependenciesValidator.cs, 5
- ./Decorators/LinksDisposableDecoratorBase.cs, 6
- ./Decorators/LinksInnerReferenceValidator.cs, 6
- ./Decorators/LinksNonExistentReferencesCreator.cs, 7
- ./Decorators/LinksNullToSelfReferenceResolver.cs, 7
- ./Decorators/LinksSelfReferenceResolver.cs, 7
- ./Decorators/LinksUniquenessResolver.cs, 8
- ./Decorators/LinksUniquenessValidator.cs, 8
- ./Decorators/NonNullContentsLinkDeletionResolver.cs, 8
- ./Decorators/UInt64Links.cs, 9
- ./Decorators/UniLinks.cs, 10
- ./Doublet.cs, 15
- ./DoubletComparer.cs, 15
- ./Hybrid.cs, 16
- ./ILinks.cs, 17
- ./ILinksExtensions.cs, 17
- ./ISynchronizedLinks.cs, 27
- ./Incrementers/FrequencyIncrementer.cs, 26
- ./Incrementers/LinkFrequencyIncrementer.cs, 26
- ./Incrementers/UnaryNumberIncrementer.cs, 27
- ./Link.cs, 27
- ./LinkExtensions.cs, 30
- ./LinksOperatorBase.cs, 30
- ./PropertyOperators/DefaultLinkPropertyOperator.cs, 30
- ./PropertyOperators/FrequencyPropertyOperator.cs, 31
- ./ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs, 40
- ./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs, 41
- ./ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 32
- ./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs, 54
- ./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs, 54
- ./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 47
- ./Sequences/Converters/BalancedVariantConverter.cs, 61
- ./Sequences/Converters/CompressingConverter.cs, 62
- ./Sequences/Converters/LinksListToSequenceConverterBase.cs, 65
- ./Sequences/Converters/OptimalVariantConverter.cs, 65
- ./Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs, 66
- ./Sequences/CriteriaMatchers/DefaultSequenceElementCriteriaMatcher.cs, 67
- ./Sequences/CriteriaMatchers/MarkedSequenceCriteriaMatcher.cs, 67
- ./Sequences/DefaultSequenceAppender.cs, 67
- ./Sequences/DuplicateSegmentsCounter.cs, 68
- ./Sequences/DuplicateSegmentsProvider.cs, 68
- ./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs, 70
- ./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToItsFrequencyNumberConverter.cs, 71
- ./Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 71
- ./Sequences/Frequencies/Cache/LinkFrequency.cs, 73
- ./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 73
- ./Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 73
- ./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 74
- ./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 74
- ./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 75
- ./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 75
- ./Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 76
- ./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 76
- ./Sequences/HeightProviders/ISequenceHeightProvider.cs, 77
- ./Sequences/Sequences.Experiments.ReadSequence.cs, 112
- ./Sequences/Sequences.Experiments.cs, 86
- ./Sequences/Sequences.cs, 77
- ./Sequences/SequencesExtensions.cs, 114

- ./Sequences/SequencesIndexer.cs, 114
- ./Sequences/SequencesOptions.cs, 115
- ./Sequences/UnicodeMap.cs, 117
- ./Sequences/Walkers/LeftSequenceWalker.cs, 119
- ./Sequences/Walkers/RightSequenceWalker.cs, 119
- ./Sequences/Walkers/SequenceWalkerBase.cs, 120
- ./Stacks/Stack.cs, 121
- ./Stacks/StackExtensions.cs, 121
- ./SynchronizedLinks.cs, 122
- ./UInt64Link.cs, 122
- ./UInt64LinkExtensions.cs, 125
- ./UInt64LinksExtensions.cs, 125
- ./UInt64LinksTransactionsLayer.cs, 127