

# LinksPlatform's Platform.Data.Doublets Class Library

## ./Converters/AddressToUnaryNumberConverter.cs

```

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

```

## ./Converters/LinkToItsFrequencyNumberConveter.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Converters
6  {
7      public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
8          ⇨ IConverter<Doublet<TLink>, TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ⇨ EqualityComparer<TLink>.Default;
12
13         private readonly ISpecificPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
14         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
15
16         public LinkToItsFrequencyNumberConveter(
17             ILinks<TLink> links,
18             ISpecificPropertyOperator<TLink, TLink> frequencyPropertyOperator,
19             IConverter<TLink> unaryNumberToAddressConverter)
20             : base(links)
21         {
22             _frequencyPropertyOperator = frequencyPropertyOperator;
23             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
24         }
25
26         public TLink Convert(Doublet<TLink> doublet)
27         {
28             var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
29             if (_equalityComparer.Equals(link, Links.Constants.Null))
30             {
31                 throw new ArgumentException($"Link with {doublet.Source} source and {doublet.Target}
32                     ⇨ target not found.", nameof(doublet));
33             }
34             var frequency = _frequencyPropertyOperator.Get(link);
35             if (_equalityComparer.Equals(frequency, default))
36             {
37                 return default;
38             }
39             var frequencyNumber = Links.GetSource(frequency);
40             var number = _unaryNumberToAddressConverter.Convert(frequencyNumber);
41             return number;
42         }
43     }
44 }

```

```

40     }
41 }

```

# ./Converters/PowerOf2ToUnaryNumberConverter.cs

```

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

```

# ./Converters/UnaryNumberToAddressAddOperationConverter.cs

```

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

```

```

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

```

#### ./Converters/UnaryNumberToAddressOrOperationConverter.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Reflection;
4  using Platform.Numbers;
5
6  namespace Platform.Data.Doublets.Converters
7  {
8      public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
9          ⇨ IConverter<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ⇨ EqualityComparer<TLink>.Default;
13
14         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
15
16         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int, TLink>
17             ⇨ powerOf2ToUnaryNumberConverter)
18             : base(links)
19         {
20             _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
21             for (int i = 0; i < CachedTypeInfo<TLink>.BitsLength; i++)
22             {
23                 _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
24             }
25
26             public TLink Convert(TLink sourceNumber)
27             {
28                 var source = sourceNumber;
29                 var target = Links.Constants.Null;
30                 while (!_equalityComparer.Equals(source, Links.Constants.Null))
31                 {
32                     if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
33                     {
34                         source = Links.Constants.Null;
35                     }
36                     else
37                     {
38                         powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
39                         source = Links.GetTarget(source);
40                     }
41                     target = (Integer<TLink>)((Integer<TLink>)target | 1UL << powerOf2Index); //
42                     ⇨ MathHelpers.Or(target, MathHelpers.ShiftLeft(One, powerOf2Index))
43                 }
44                 return target;
45             }
46         }
47     }
48 }

```

#### ./Decorators/LinksCascadeDependenciesResolver.cs

```

1  using System.Collections.Generic;
2  using Platform.Collections.Arrays;
3  using Platform.Numbers;
4
5  namespace Platform.Data.Doublets.Decorators
6  {
7      public class LinksCascadeDependenciesResolver<TLink> : LinksDecoratorBase<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ⇨ EqualityComparer<TLink>.Default;
11
12         public LinksCascadeDependenciesResolver(ILinks<TLink> links) : base(links) { }
13     }
14 }

```

```

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

```

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

```

#### ./Decorators/LinksInnerReferenceValidator.cs

```

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

```

```

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

```

#### ./Decorators/LinksNonExistentReferencesCreator.cs

```

1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Decorators
4 {
5     /// <remarks>
6     /// Not practical if newSource and newTarget are too big.
7     /// To be able to use practical version we should allow to create link at any specific location
8     ↪ inside ResizableDirectMemoryLinks.
9     /// This in turn will require to implement not a list of empty links, but a list of ranges to store
10    ↪ 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], restrictions[Constants.TargetPart]);
19            return base.Update(restrictions);
20        }
21    }
22 }

```

#### ./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 }

```

```

22     }
23 }
24 }

```

#### ./Decorators/LinksSelfReferenceResolver.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets.Decorators
5  {
6      public class LinksSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
7      {
8          private static readonly EqualityComparer<TLink> _equalityComparer =
9              ↳ EqualityComparer<TLink>.Default;
10
11          public LinksSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
12
13          public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
14          {
15              if (!_equalityComparer.Equals(Constants.Any, Constants.Itself)
16                  && ((restrictions.Count > Constants.IndexPart) &&
17                      ↳ _equalityComparer.Equals(restrictions[Constants.IndexPart], Constants.Itself))
18                  || ((restrictions.Count > Constants.SourcePart) &&
19                      ↳ _equalityComparer.Equals(restrictions[Constants.SourcePart], Constants.Itself))
20                  || ((restrictions.Count > Constants.TargetPart) &&
21                      ↳ _equalityComparer.Equals(restrictions[Constants.TargetPart], Constants.Itself))))
22              {
23                  return Constants.Continue;
24              }
25              return base.Each(handler, restrictions);
26          }
27
28          public override TLink Update(IList<TLink> restrictions)
29          {
30              restrictions[Constants.SourcePart] =
31                  ↳ _equalityComparer.Equals(restrictions[Constants.SourcePart], Constants.Itself) ?
32                  ↳ restrictions[Constants.IndexPart] : restrictions[Constants.SourcePart];
33              restrictions[Constants.TargetPart] =
34                  ↳ _equalityComparer.Equals(restrictions[Constants.TargetPart], Constants.Itself) ?
35                  ↳ restrictions[Constants.IndexPart] : restrictions[Constants.TargetPart];
36              return base.Update(restrictions);
37          }
38      }
39  }
40
41 }

```

#### ./Decorators/LinksUniquenessResolver.cs

```

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

```

#### ./Decorators/LinksUniquenessValidator.cs

```

1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      public class LinksUniquenessValidator<T> : LinksDecoratorBase<T>
6      {
7          public LinksUniquenessValidator(ILinks<T> links) : base(links) { }

```

```

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

```

#### ./Decorators/NonNullContentsLinkDeletionResolver.cs

```

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

```

#### ./Decorators/UInt64Links.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Collections;
4 using Platform.Collections.Arrays;
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     /// <summary>
9     /// Представляет объект для работы с базой данных (файлом) в формате Links (массива взаимосвязей).
10    /// </summary>
11    /// <remarks>
12    /// Возможные оптимизации:
13    /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
14    /// + меньше объём БД
15    /// - меньше производительность
16    /// - больше ограничение на количество связей в БД)
17    /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
18    /// + меньше объём БД
19    /// - больше сложность
20    ///
21    /// AVL - высота дерева может позволить точно расчитать размер дерева, нет необходимости в SBT.
22    /// AVL дерево можно прошить.
23    ///
24    /// Текущее теоретическое ограничение на размер связей - long.MaxValue
25    /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами (битовыми
26    ↪ строками) - вариант матрицы (выстраиваемой лениво).
27    ///
28    /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут выбрасываться
29    ↪ только при #if DEBUG
30    /// </remarks>
31    public class UInt64Links : LinksDisposableDecoratorBase<ulong>
32    {
33        public UInt64Links(ILinks<ulong> links) : base(links) { }
34
35        public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
36        {
37            this.EnsureLinkIsAnyOrExists(restrictions);
38            return Links.Each(handler, restrictions);
39        }
40
41        public override ulong Create() => Links.CreatePoint();
42
43        public override ulong Update(IList<ulong> restrictions)
44        {
45            if (restrictions.IsNullOrEmpty())
46            {
47                return Constants.Null;
48            }
49            // TODO: Remove usages of these hacks (these should not be backwards compatible)
50            if (restrictions.Count == 2)
51            {
52                return this.Merge(restrictions[0], restrictions[1]);
53            }
54            if (restrictions.Count == 4)
55            {
56                return this.UpdateOrCreateOrGet(restrictions[0], restrictions[1], restrictions[2],
57                ↪ restrictions[3]);
58            }
59            // TODO: Looks like this is a common type of exceptions linked with restrictions support
60            if (restrictions.Count != 3)
61            {

```



```

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

```

## ./Decorators/UniLinks.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Platform.Collections;
5  using Platform.Collections.Arrays;
6  using Platform.Collections.Lists;
7  using Platform.Helpers.Scopes;
8  using Platform.Data.Constants;
9  using Platform.Data.Universal;
10 using System.Collections.ObjectModel;
11
12 namespace Platform.Data.Doublets.Decorators
13 {
14     /// <remarks>
15     /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
16     /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed by
17     ↪ 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 IDoubletLinks
21     ↪ and ILinks.)
22     /// </remarks>
23     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
24     {

```

```

22 private static readonly EqualityComparer<TLink> _equalityComparer =
    ↳ EqualityComparer<TLink>.Default;
23
24 public UniLinks(ILinks<TLink> links) : base(links) { }
25
26 private struct Transition
27 {
28     public IList<TLink> Before;
29     public IList<TLink> After;
30
31     public Transition(IList<TLink> before, IList<TLink> after)
32     {
33         Before = before;
34         After = after;
35     }
36 }
37
38 public static readonly TLink NullConstant = Use<LinksCombinedConstants<TLink, TLink,
    ↳ int>>.Single.Null;
39 public static readonly IReadOnlyList<TLink> NullLink = new ReadOnlyCollection<TLink>(new
    ↳ List<TLink> { NullConstant, NullConstant, NullConstant });
40
41 // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
    ↳ (Links-Expression)
42 public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
    ↳ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
    ↳ substitutedHandler)
43 {
44     ///List<Transition> transitions = null;
45     ///if (!restriction.IsNullOrEmpty())
46     ///{
47     ///    // Есть причина делать проход (чтение)
48     ///    if (matchedHandler != null)
49     ///    {
50     ///        if (!substitution.IsNullOrEmpty())
51     ///        {
52     ///            // restriction => { 0, 0, 0 } | { 0 } // Create
53     ///            // substitution => { itself, 0, 0 } | { itself, itself, itself } // Create
54     ///            / Update
55     ///            // substitution => { 0, 0, 0 } | { 0 } // Delete
56     ///            transitions = new List<Transition>();
57     ///            if (Equals(substitution[Constants.IndexPart], Constants.Null))
58     ///            {
59     ///                // If index is Null, that means we always ignore every other value
60     ///                ↳ (they are also Null by definition)
61     ///                var matchDecision = matchedHandler(, NullLink);
62     ///                if (Equals(matchDecision, Constants.Break))
63     ///                {
64     ///                    return false;
65     ///                }
66     ///                if (!Equals(matchDecision, Constants.Skip))
67     ///                {
68     ///                    transitions.Add(new Transition(matchedLink, newValue));
69     ///                }
70     ///            }
71     ///            else
72     ///            {
73     ///                Func<T, bool> handler;
74     ///                handler = link =>
75     ///                {
76     ///                    var matchedLink = Memory.GetLinkValue(link);
77     ///                    var newValue = Memory.GetLinkValue(link);
78     ///                    newValue[Constants.IndexPart] = Constants.Itself;
79     ///                    newValue[Constants.SourcePart] =
80     ///                    ↳ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
81     ///                    ↳ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
82     ///                    newValue[Constants.TargetPart] =
83     ///                    ↳ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
84     ///                    ↳ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
85     ///                    var matchDecision = matchedHandler(matchedLink, newValue);
86     ///                    if (Equals(matchDecision, Constants.Break))
87     ///                    {
88     ///                        return false;
89     ///                    }
90     ///                    if (!Equals(matchDecision, Constants.Skip))
91     ///                    {
92     ///                        transitions.Add(new Transition(matchedLink, newValue));
93     ///                    }
94     ///                    return true;
95     ///                };
96     ///                if (!Memory.Each(handler, restriction))
97     ///                {
98     ///                    return Constants.Break;
99     ///                }
100            }
101        }
102    }
103    else
104    {
105        Func<T, bool> handler = link =>
106        {
107            var matchedLink = Memory.GetLinkValue(link);
108            var matchDecision = matchedHandler(matchedLink, matchedLink);
109            return !Equals(matchDecision, Constants.Break);
110        };
111        if (!Memory.Each(handler, restriction))
112        {
113            return Constants.Break;
114        }
115    }
116 }

```

```

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             ////            {
111                 ////                return Constants.Break;
112             }
113             ////            else
114             ////            {
115                 ////                return Constants.Continue;
116             }
117         }
118     }
119     ////if (substitution != null)
120     ////if {
121         ////    // Есть причина делать замену (запись)
122         ////    if (substitutedHandler != null)
123         ////    {
124             ////    }
125         ////    else
126         ////    {
127             ////    }
128         ////}
129     ////return Constants.Continue;
130
131     ///if (restriction.IsNullOrEmpty()) // Create
132     ///{
133         ///    substitution[Constants.IndexPart] = Memory.AllocateLink();
134         ///    Memory.SetLinkValue(substitution);
135     ///}
136     ///else if (substitution.IsNullOrEmpty()) // Delete
137     ///{
138         ///    Memory.FreeLink(restriction[Constants.IndexPart]);
139     ///}
140     ///else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
141     ///{
142         ///    // No need to collect links to list
143         ///    // Skip == Continue
144         ///    // No need to check substitutedHandler
145         ///    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
146         ↪ Constants.Break), restriction))
147         ///    {
148             ///        return Constants.Break;
149         ///    }
150     ///}
151     ///else // Update
152     ///{
153         ///    //List<IList<T>> matchedLinks = null;
154         ///    if (matchedHandler != null)
155         ///    {
156             ///        matchedLinks = new List<IList<T>>();
157             ///        Func<T, bool> handler = link =>
158             ///        {
159                 ///            var matchedLink = Memory.GetLinkValue(link);
160                 ///            var matchDecision = matchedHandler(matchedLink);
161                 ///            if (Equals(matchDecision, Constants.Break))
162                 ///                return false;
163                 ///            if (!Equals(matchDecision, Constants.Skip))
164                 ///                matchedLinks.Add(matchedLink);
165                 ///            return true;
166             ///        };
167             ///        if (!Memory.Each(handler, restriction))
168             ///        {
169                 ///            return Constants.Break;
170             }
171             ///        if (!matchedLinks.IsNullOrEmpty())
172             ///        {
173                 ///            var totalMatchedLinks = matchedLinks.Count;
174                 ///            for (var i = 0; i < totalMatchedLinks; i++)
175                 ///            {
176                     ///                var matchedLink = matchedLinks[i];
177                     ///                if (substitutedHandler != null)
178                     ///                {
179                         ///                    var newValue = new List<T>(); // TODO: Prepare value to update here
180                         ///                    // TODO: Decide is it actually needed to use Before and After
181                         ↪ substitution handling.
182                         ///                    var substitutedDecision = substitutedHandler(matchedLink, newValue);
183                         ///                    if (Equals(substitutedDecision, Constants.Break))
184                         ///                        return Constants.Break;
185                         ///                    if (Equals(substitutedDecision, Constants.Continue))

```

```

180         {
181             // Actual update here
182             Memory.SetLinkValue(newValue);
183         }
184         if (Equals(substitutedDecision, Constants.Skip))
185         {
186             // Cancel the update. TODO: decide use separate Cancel constant or
187             // Skip is enough?
188             }
189         }
190     }
191 }
192 return Constants.Continue;
193 }
194
195 public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink> matchHandler,
196     → IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink> substitutionHandler)
197 {
198     if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
199     {
200         return Constants.Continue;
201     }
202     else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO: Check if it
203     → is a correct condition
204     {
205         // Or it only applies to trigger without matchHandler.
206         throw new NotImplementedException();
207     }
208     else if (!substitution.IsNullOrEmpty()) // Creation
209     {
210         var before = ArrayPool<TLink>.Empty;
211         // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
212         → (пройти мимо) или пустить (взять)?
213         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
214             → Constants.Break))
215         {
216             return Constants.Break;
217         }
218         var after = (IList<TLink>)substitution.ToArray();
219         if (_equalityComparer.Equals(after[0], default))
220         {
221             var newLink = Links.Create();
222             after[0] = newLink;
223         }
224         if (substitution.Count == 1)
225         {
226             after = Links.GetLink(substitution[0]);
227         }
228         else if (substitution.Count == 3)
229         {
230             Links.Update(after);
231         }
232         else
233         {
234             throw new NotSupportedException();
235         }
236         if (matchHandler != null)
237         {
238             return substitutionHandler(before, after);
239         }
240         return Constants.Continue;
241     }
242     else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
243     {
244         if (patternOrCondition.Count == 1)
245         {
246             var linkToDelete = patternOrCondition[0];
247             var before = Links.GetLink(linkToDelete);
248             if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
249                 → Constants.Break))
250             {
251                 return Constants.Break;
252             }
253             var after = ArrayPool<TLink>.Empty;
254             Links.Update(linkToDelete, Constants.Null, Constants.Null);
255             Links.Delete(linkToDelete);
256             if (matchHandler != null)
257             {
258                 return substitutionHandler(before, after);
259             }
260             return Constants.Continue;
261         }
262         else
263         {
264             throw new NotSupportedException();
265         }
266     }
267 }

```



```

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)
18         ↪ && _equalityComparer.Equals(x.Target, y.Target);
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     public int GetHashCode(Doublet<T> obj) => unchecked(obj.Source.GetHashCode() << 15 ^
22         ↪ obj.Target.GetHashCode());

```

./Doublet.cs

```

1     using System;
2     using System.Collections.Generic;
3
4     namespace Platform.Data.Doublets
5     {
6         public struct Doublet<T> : IEquatable<Doublet<T>>
7         {
8             private static readonly EqualityComparer<T> _equalityComparer = EqualityComparer<T>.Default;
9
10            public T Source { get; set; }
11            public T Target { get; set; }
12
13            public Doublet(T source, T target)
14            {
15                Source = source;
16                Target = target;
17            }
18
19            public override string ToString() => $"{Source}->{Target}";
20
21            public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source) &&
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 = typeof(MathHelpers).GetTypeInfo().GetMethod("Abs").MakeGenericMethod(signedType);
34                var negate =
35                    ↪ typeof(MathHelpers).GetTypeInfo().GetMethod("Negate").MakeGenericMethod(signedType);
36                var absoluteValue = abs.Invoke(null, new[] { signedValue });
37                var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) : absoluteValue;
38                Value = To.UnsignedAs<T>(resultValue);
39            }
40
41            public static implicit operator Hybrid<T>(T integer) => new Hybrid<T>(integer);
42
43            public static explicit operator Hybrid<T>(ulong integer) => new Hybrid<T>(integer);
44
45            public static explicit operator Hybrid<T>(long integer) => new Hybrid<T>(integer);
46
47            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) => Convert.ToUInt64(hybrid.Value);
54     public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;
55     public static explicit operator uint(Hybrid<T> hybrid) => Convert.ToUInt32(hybrid.Value);
56     public static explicit operator int(Hybrid<T> hybrid) => Convert.ToInt32(hybrid.AbsoluteValue);
57     public static explicit operator ushort(Hybrid<T> hybrid) => Convert.ToUInt16(hybrid.Value);
58     public static explicit operator short(Hybrid<T> hybrid) =>
59         ↪ Convert.ToInt16(hybrid.AbsoluteValue);
60     public static explicit operator byte(Hybrid<T> hybrid) => Convert.ToByte(hybrid.Value);
61     public static explicit operator sbyte(Hybrid<T> hybrid) =>
62         ↪ Convert.ToSByte(hybrid.AbsoluteValue);
63     public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
64         ↪ default(T).ToString() : IsExternal ? $"{<AbsoluteValue>}" : Value.ToString();
65 }
66 }
67 }
68 }
69 }
70 }
71 }
72 }
73 }
74 }
75 }
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 :
42                 ↪ (Integer<TLink>)links.Count() - (ulong)amountOfDeletions;
43             for (long i = 0; i < amountOfDeletions; i++)

```

```

43     {
44         var linksAddressRange = new Range<ulong>(min, (Integer<TLink>)links.Count());
45         Integer<TLink> link = RandomHelpers.Default.NextUInt64(linksAddressRange);
46         links.Delete(link);
47         if ((Integer<TLink>)links.Count() < min)
48         {
49             break;
50         }
51     }
52 }
53
54 /// <remarks>
55 /// TODO: Возможно есть очень простой способ это сделать.
56 /// (Например просто удалить файл, или изменить его размер таким образом,
57 /// чтобы удалился весь контент)
58 /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
59 /// </remarks>
60 public static void DeleteAll<TLink>(this ILinks<TLink> links)
61 {
62     var equalityComparer = EqualityComparer<TLink>.Default;
63     var comparer = Comparer<TLink>.Default;
64     for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
        ↪ ArithmeticHelpers.Decrement(i))
65     {
66         links.Delete(i);
67         if (!equalityComparer.Equals(links.Count(), ArithmeticHelpers.Decrement(i)))
68         {
69             i = links.Count();
70         }
71     }
72 }
73
74 public static TLink First<TLink>(this ILinks<TLink> links)
75 {
76     TLink firstLink = default;
77     var equalityComparer = EqualityComparer<TLink>.Default;
78     if (equalityComparer.Equals(links.Count(), default))
79     {
80         throw new Exception("В хранилище нет связей.");
81     }
82     links.Each(links.Constants.Any, links.Constants.Any, link =>
83     {
84         firstLink = link[links.Constants.IndexPart];
85         return links.Constants.Break;
86     });
87     if (equalityComparer.Equals(firstLink, default))
88     {
89         throw new Exception("В процессе поиска по хранилищу не было найдено связей.");
90     }
91     return firstLink;
92 }
93
94 public static bool IsInnerReference<TLink>(this ILinks<TLink> links, TLink reference)
95 {
96     var constants = links.Constants;
97     var comparer = Comparer<TLink>.Default;
98     return comparer.Compare(constants.MinPossibleIndex, reference) >= 0 &&
        ↪ comparer.Compare(reference, constants.MaxPossibleIndex) <= 0;
99 }
100
101 #region Paths
102
103 /// <remarks>
104 /// TODO: Как так? Как то что ниже может быть корректно?
105 /// Скорее всего практически не применимо
106 /// Предполагалось, что можно было конвертировать формируемый в проходе через SequenceWalker
107 /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
108 /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
109 /// </remarks>
110 public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[] path)
111 {
112     var current = path[0];
113     //EnsureLinkExists(current, "path");
114     if (!links.Exists(current))
115     {
116         return false;
117     }
118     var equalityComparer = EqualityComparer<TLink>.Default;
119     var constants = links.Constants;
120     for (var i = 1; i < path.Length; i++)
121     {
122         var next = path[i];
123         var values = links.GetLink(current);
124         var source = values[constants.SourcePart];
125         var target = values[constants.TargetPart];
126         if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source, next))
127         {

```



```

128         //throw new Exception(string.Format("Невозможно выбрать путь, так как и Source и
129         ↪ Target совпадают с элементом пути {0}.", next));
130         return false;
131     }
132     if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next, target))
133     {
134         //throw new Exception(string.Format("Невозможно продолжить путь через элемент пути
135         ↪ {0}", next));
136         return false;
137     }
138     current = next;
139 }
140 return true;
141 }
142
143 /// <remarks>
144 /// Может потребовать дополнительного стека для PathElement's при использовании SequenceWalker.
145 /// </remarks>
146 public static TLink GetByKeyes<TLink>(this ILinks<TLink> links, TLink root, params int[] path)
147 {
148     links.EnsureLinkExists(root, "root");
149     var currentLink = root;
150     for (var i = 0; i < path.Length; i++)
151     {
152         currentLink = links.GetLink(currentLink)[path[i]];
153     }
154     return currentLink;
155 }
156
157 public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink> links,
158 ↪ TLink root, ulong size, ulong index)
159 {
160     var constants = links.Constants;
161     var source = constants.SourcePart;
162     var target = constants.TargetPart;
163     if (!MathHelpers.IsPowerOfTwo(size))
164     {
165         throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other than
166         ↪ powers of two are not supported.");
167     }
168     var path = new BitArray(BitConverter.GetBytes(index));
169     var length = BitwiseHelpers.GetLowestBitPosition(size);
170     links.EnsureLinkExists(root, "root");
171     var currentLink = root;
172     for (var i = length - 1; i >= 0; i--)
173     {
174         currentLink = links.GetLink(currentLink)[path[i] ? target : source];
175     }
176     return currentLink;
177 }
178
179 #endregion
180
181 /// <summary>
182 /// Возвращает индекс указанной связи.
183 /// </summary>
184 /// <param name="links">Хранилище связей.</param>
185 /// <param name="link">Связь представленная списком, состоящим из её адреса и
186 ↪ содержимого.</param>
187 /// <returns>Индекс начальной связи для указанной связи.</returns>
188 [MethodImpl(MethodImplOptions.AggressiveInlining)]
189 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
190 ↪ link[links.Constants.IndexPart];
191
192 /// <summary>
193 /// Возвращает индекс начальной (Source) связи для указанной связи.
194 /// </summary>
195 /// <param name="links">Хранилище связей.</param>
196 /// <param name="link">Индекс связи.</param>
197 /// <returns>Индекс начальной связи для указанной связи.</returns>
198 [MethodImpl(MethodImplOptions.AggressiveInlining)]
199 public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
200 ↪ links.GetLink(link)[links.Constants.SourcePart];
201
202 /// <summary>
203 /// Возвращает индекс начальной (Source) связи для указанной связи.
204 /// </summary>
205 /// <param name="links">Хранилище связей.</param>
206 /// <param name="link">Связь представленная списком, состоящим из её адреса и
207 ↪ содержимого.</param>
208 /// <returns>Индекс начальной связи для указанной связи.</returns>
209 [MethodImpl(MethodImplOptions.AggressiveInlining)]
210 public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
211 ↪ link[links.Constants.SourcePart];
212
213 /// <summary>
214 /// Возвращает индекс конечной (Target) связи для указанной связи.

```

```

206 /// </summary>
207 /// <param name="links">Хранилище связей.</param>
208 /// <param name="link">Индекс связи.</param>
209 /// <returns>Индекс конечной связи для указанной связи.</returns>
210 [MethodImpl(MethodImplOptions.AggressiveInlining)]
211 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
    ↳ links.GetLink(link)[links.Constants.TargetPart];

212
213 /// <summary>
214 /// Возвращает индекс конечной (Target) связи для указанной связи.
215 /// </summary>
216 /// <param name="links">Хранилище связей.</param>
217 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
218 /// <returns>Индекс конечной связи для указанной связи.</returns>
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.TargetPart];

221
222 /// <summary>
223 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик (handler) для
    ↳ каждой подходящей связи.
224 /// </summary>
225 /// <param name="links">Хранилище связей.</param>
226 /// <param name="handler">Обработчик каждой подходящей связи.</param>
227 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение может иметь
    ↳ значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту, Any - отсутствие
    ↳ ограничения, 1..∞ конкретный адрес связи.</param>
228 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
229 [MethodImpl(MethodImplOptions.AggressiveInlining)]
230 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink> handler,
    ↳ params TLink[] restrictions)
231     => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
    ↳ links.Constants.Continue);

232
233 /// <summary>
234 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик (handler) для
    ↳ каждой подходящей связи.
235 /// </summary>
236 /// <param name="links">Хранилище связей.</param>
237 /// <param name="source">Значение, определяющее соответствующие шаблону связи. (Constants.Null
    ↳ - 0-я связь, обозначающая ссылку на пустоту в качестве начала, Constants.Any - любое
    ↳ начало, 1..∞ конкретное начало)</param>
238 /// <param name="target">Значение, определяющее соответствующие шаблону связи. (Constants.Null
    ↳ - 0-я связь, обозначающая ссылку на пустоту в качестве конца, Constants.Any - любой конец,
    ↳ 1..∞ конкретный конец)</param>
239 /// <param name="handler">Обработчик каждой подходящей связи.</param>
240 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
241 [MethodImpl(MethodImplOptions.AggressiveInlining)]
242 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<TLink, bool> handler)
243 {
244     var constants = links.Constants;
245     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
    ↳ constants.Break, constants.Any, source, target);
246 }

247
248 /// <summary>
249 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик (handler) для
    ↳ каждой подходящей связи.
250 /// </summary>
251 /// <param name="links">Хранилище связей.</param>
252 /// <param name="source">Значение, определяющее соответствующие шаблону связи. (Constants.Null
    ↳ - 0-я связь, обозначающая ссылку на пустоту в качестве начала, Constants.Any - любое
    ↳ начало, 1..∞ конкретное начало)</param>
253 /// <param name="target">Значение, определяющее соответствующие шаблону связи. (Constants.Null
    ↳ - 0-я связь, обозначающая ссылку на пустоту в качестве конца, Constants.Any - любой конец,
    ↳ 1..∞ конкретный конец)</param>
254 /// <param name="handler">Обработчик каждой подходящей связи.</param>
255 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
256 [MethodImpl(MethodImplOptions.AggressiveInlining)]
257 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<IList<TLink>, TLink> handler)
258 {
259     var constants = links.Constants;
260     return links.Each(handler, constants.Any, source, target);
261 }

262
263 [MethodImpl(MethodImplOptions.AggressiveInlining)]
264 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ restrictions)
265 {

```

```

266     var constants = links.Constants;
267     int listSize = (Integer<TLink>)links.Count(restrictions);
268     var list = new IList<TLink>[listSize];
269     if (listSize > 0)
270     {
271         var filler = new ArrayFiller<IList<TLink>, TLink>(list, links.Constants.Continue);
272         links.Each(filler.AddAndReturnConstant, restrictions);
273     }
274     return list;
275 }
276
277 /// <summary>
278 /// Возвращает значение, определяющее существует ли связь с указанными началом и концом в
279   ↳ хранилище связей.
280 /// </summary>
281 /// <param name="links">Хранилище связей.</param>
282 /// <param name="source">Начало связи.</param>
283 /// <param name="target">Конец связи.</param>
284 /// <returns>Значение, определяющее существует ли связь.</returns>
285 [MethodImpl(MethodImplOptions.AggressiveInlining)]
286 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target) =>
287   ↳ Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target), default)
288   ↳ > 0;
289
290 #region Ensure
291 // TODO: May be move to EnsureExtensions or make it both there and here
292
293 [MethodImpl(MethodImplOptions.AggressiveInlining)]
294 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink reference,
295   ↳ string argumentName)
296 {
297     if (links.IsInnerReference(reference) && !links.Exists(reference))
298     {
299         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
300     }
301 }
302
303 [MethodImpl(MethodImplOptions.AggressiveInlining)]
304 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, IList<TLink>
305   ↳ restrictions, string argumentName)
306 {
307     for (int i = 0; i < restrictions.Count; i++)
308     {
309         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
310     }
311 }
312
313 [MethodImpl(MethodImplOptions.AggressiveInlining)]
314 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
315   ↳ restrictions)
316 {
317     for (int i = 0; i < restrictions.Count; i++)
318     {
319         links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
320     }
321 }
322
323 [MethodImpl(MethodImplOptions.AggressiveInlining)]
324 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link, string
325   ↳ argumentName)
326 {
327     var equalityComparer = EqualityComparer<TLink>.Default;
328     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
329     {
330         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
331     }
332 }
333
334 [MethodImpl(MethodImplOptions.AggressiveInlining)]
335 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink link,
336   ↳ string argumentName)
337 {
338     var equalityComparer = EqualityComparer<TLink>.Default;
339     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
340     {
341         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
342     }
343 }
344
345 /// <param name="links">Хранилище связей.</param>
346 [MethodImpl(MethodImplOptions.AggressiveInlining)]
347 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source, TLink
348   ↳ target)
349 {
350     if (links.Exists(source, target))
351     {
352

```

```

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) =>
358     ↪ links.EnsureCreated(links.Create, addresses);
359
360 /// <param name="links">Хранилище связей.</param>
361 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
362     ↪ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
363
364 /// <param name="links">Хранилище связей.</param>
365 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator, params
366     ↪ TLink[] addresses)
367 {
368     var constants = links.Constants;
369     var nonExistentAddresses = new HashSet<ulong>(addresses.Where(x =>
370     ↪ !links.Exists(x)).Select(x => (ulong)(Integer<TLink>)x));
371     if (nonExistentAddresses.Count > 0)
372     {
373         var max = nonExistentAddresses.Max();
374         // TODO: Эту верхнюю границу нужно разрешить переопределять (проверить применяется ли
375         ↪ эта логика)
376         max = Math.Min(max, (Integer<TLink>)constants.MaxPossibleIndex);
377         var createdLinks = new List<TLink>();
378         var equalityComparer = EqualityComparer<TLink>.Default;
379         TLink createdLink = creator();
380         while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
381         {
382             createdLinks.Add(createdLink);
383         }
384         for (var i = 0; i < createdLinks.Count; i++)
385         {
386             if (!nonExistentAddresses.Contains((Integer<TLink>)createdLinks[i]))
387             {
388                 links.Delete(createdLinks[i]);
389             }
390         }
391     }
392 }
393
394 #endregion
395
396 /// <param name="links">Хранилище связей.</param>
397 public static ulong DependenciesCount<TLink>(this ILinks<TLink> links, TLink link)
398 {
399     var constants = links.Constants;
400     var values = links.GetLink(link);
401     ulong referencesAsSource = (Integer<TLink>)links.Count(constants.Any, link, constants.Any);
402     var equalityComparer = EqualityComparer<TLink>.Default;
403     if (equalityComparer.Equals(values[constants.SourcePart], link))
404     {
405         referencesAsSource--;
406     }
407     ulong referencesAsTarget = (Integer<TLink>)links.Count(constants.Any, constants.Any, link);
408     if (equalityComparer.Equals(values[constants.TargetPart], link))
409     {
410         referencesAsTarget--;
411     }
412     return referencesAsSource + referencesAsTarget;
413 }
414
415 /// <param name="links">Хранилище связей.</param>
416 [MethodImpl(MethodImplOptions.AggressiveInlining)]
417 public static bool DependenciesExist<TLink>(this ILinks<TLink> links, TLink link) =>
418     ↪ links.DependenciesCount(link) > 0;
419
420 /// <param name="links">Хранилище связей.</param>
421 [MethodImpl(MethodImplOptions.AggressiveInlining)]
422 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source, TLink
423     ↪ target)
424 {
425     var constants = links.Constants;
426     var values = links.GetLink(link);
427     var equalityComparer = EqualityComparer<TLink>.Default;
428     return equalityComparer.Equals(values[constants.SourcePart], source) &&
429     ↪ equalityComparer.Equals(values[constants.TargetPart], target);

```

```

422 }
423
424 /// <summary>
425 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
426 /// </summary>
427 /// <param name="links">Хранилище связей.</param>
428 /// <param name="source">Индекс связи, которая является началом для искомой связи.</param>
429 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
430 /// <returns>Индекс искомой связи с указанными Source (началом) и Target (концом).</returns>
431 [MethodImpl(MethodImplOptions.AggressiveInlining)]
432 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink target)
433 {
434     var contents = links.Constants;
435     var setter = new Setter<TLink, TLink>(contents.Continue, contents.Break, default);
436     links.Each(setter.SetFirstAndReturnFalse, contents.Any, source, target);
437     return setter.Result;
438 }
439
440 /// <param name="links">Хранилище связей.</param>
441 [MethodImpl(MethodImplOptions.AggressiveInlining)]
442 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
443 {
444     var link = links.Create();
445     return links.Update(link, link, link);
446 }
447
448 /// <param name="links">Хранилище связей.</param>
449 [MethodImpl(MethodImplOptions.AggressiveInlining)]
450 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
451     ↪ target) => links.Update(links.Create(), source, target);
452
453 /// <summary>
454 /// Обновляет связь с указанными началом (Source) и концом (Target)
455 /// на связь с указанными началом (NewSource) и концом (NewTarget).
456 /// </summary>
457 /// <param name="links">Хранилище связей.</param>
458 /// <param name="link">Индекс обновляемой связи.</param>
459 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
460     ↪ выполняется обновление.</param>
461 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую выполняется
462     ↪ обновление.</param>
463 /// <returns>Индекс обновлённой связи.</returns>
464 [MethodImpl(MethodImplOptions.AggressiveInlining)]
465 public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource, TLink
466     ↪ newTarget) => links.Update(new[] { link, newSource, newTarget });
467
468 /// <summary>
469 /// Обновляет связь с указанными началом (Source) и концом (Target)
470 /// на связь с указанными началом (NewSource) и концом (NewTarget).
471 /// </summary>
472 /// <param name="links">Хранилище связей.</param>
473 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение может иметь
474     ↪ значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту, Itself - требование
475     ↪ установить ссылку на себя, 1..∞ конкретный адрес другой связи.</param>
476 /// <returns>Индекс обновлённой связи.</returns>
477 [MethodImpl(MethodImplOptions.AggressiveInlining)]
478 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
479 {
480     if (restrictions.Length == 2)
481     {
482         return links.Merge(restrictions[0], restrictions[1]);
483     }
484     if (restrictions.Length == 4)
485     {
486         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1], restrictions[2],
487             ↪ restrictions[3]);
488     }
489     else
490     {
491         return links.Update(restrictions);
492     }
493 }
494
495 /// <summary>
496 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи с
497     ↪ указанными Source (началом) и Target (концом).
498 /// </summary>
499 /// <param name="links">Хранилище связей.</param>
500 /// <param name="source">Индекс связи, которая является началом на создаваемой связи.</param>
501 /// <param name="target">Индекс связи, которая является концом для создаваемой связи.</param>
502 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
503 [MethodImpl(MethodImplOptions.AggressiveInlining)]
504 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink target)
505 {
506     var link = links.SearchOrDefault(source, target);
507     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
514   ↳ выполняется обновление.</param>
515 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую выполняется
516   ↳ обновление.</param>
517 /// <returns>Индекс обновлённой связи.</returns>
518 [MethodImpl(MethodImplOptions.AggressiveInlining)]
519 public static TLink UpdateOrCreateGet<TLink>(this ILinks<TLink> links, TLink source, TLink
520   ↳ target, TLink newSource, TLink newTarget)
521 {
522     var equalityComparer = EqualityComparer<TLink>.Default;
523     var link = links.SearchOrDefault(source, target);
524     if (equalityComparer.Equals(link, default))
525     {
526         return links.CreateAndUpdate(newSource, newTarget);
527     }
528     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
529   ↳ target))
530     {
531         return link;
532     }
533     return links.Update(link, newSource, newTarget);
534 }
535
536 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
537 /// <param name="links">Хранилище связей.</param>
538 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
539 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
540 [MethodImpl(MethodImplOptions.AggressiveInlining)]
541 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
542 {
543     var link = links.SearchOrDefault(source, target);
544     if (!EqualityComparer<TLink>.Default.Equals(link, default))
545     {
546         links.Delete(link);
547         return link;
548     }
549     return default;
550 }
551
552 /// <summary>Удаляет несколько связей.</summary>
553 /// <param name="links">Хранилище связей.</param>
554 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
555 [MethodImpl(MethodImplOptions.AggressiveInlining)]
556 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
557 {
558     for (int i = 0; i < deletedLinks.Count; i++)
559     {
560         links.Delete(deletedLinks[i]);
561     }
562 }
563
564 // Replace one link with another (replaced link is deleted, children are updated or deleted)
565 public static TLink Merge<TLink>(this ILinks<TLink> links, TLink linkIndex, TLink newLink)
566 {
567     var equalityComparer = EqualityComparer<TLink>.Default;
568     if (equalityComparer.Equals(linkIndex, newLink))
569     {
570         return newLink;
571     }
572     var constants = links.Constants;
573     ulong referencesAsSourceCount = (Integer<TLink>)links.Count(constants.Any, linkIndex,
574   ↳ constants.Any);
575     ulong referencesAsTargetCount = (Integer<TLink>)links.Count(constants.Any, constants.Any,
576   ↳ linkIndex);
577     var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(linkIndex)) &&
578   ↳ referencesAsSourceCount == 1 && referencesAsTargetCount == 1;
579     if (!isStandalonePoint)
580     {
581         var totalReferences = referencesAsSourceCount + referencesAsTargetCount;
582         if (totalReferences > 0)
583         {
584             var references = ArrayPool.Allocate<TLink>((long)totalReferences);

```

```

578         var referencesFiller = new ArrayFiller<TLink, TLink>(references,
579             ↪ links.Constants.Continue);
580         links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any, linkIndex,
581             ↪ constants.Any);
582         links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any,
583             ↪ constants.Any, linkIndex);
584         for (ulong i = 0; i < referencesAsSourceCount; i++)
585         {
586             var reference = references[i];
587             if (equalityComparer.Equals(reference, linkIndex))
588             {
589                 continue;
590             }
591             links.Update(reference, newLink, links.GetTarget(reference));
592         }
593         for (var i = (long)referencesAsSourceCount; i < references.Length; i++)
594         {
595             var reference = references[i];
596             if (equalityComparer.Equals(reference, linkIndex))
597             {
598                 continue;
599             }
600             links.Update(reference, links.GetSource(reference), newLink);
601         }
602         ArrayPool.Free(references);
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>, IIncrementer<IList<TLink>>
7      {
8          private readonly ISpecificPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
9          private readonly IIncrementer<TLink> _frequencyIncrementer;
10
11          public LinkFrequencyIncrementer(ILinks<TLink> links, ISpecificPropertyOperator<TLink, TLink>
12              ↪ frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
13              : base(links)
14          {
15              _frequencyPropertyOperator = frequencyPropertyOperator;
16          }
17      }
18  }

```

```

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

```

#### ./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) => _unaryOne =
14          ↪ 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, target);
64
65 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
66
67 public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
68     && _equalityComparer.Equals(Source, _constants.Null)
69     && _equalityComparer.Equals(Target, _constants.Null);
70
71 public override bool Equals(object other) => other is Link<TLink> && Equals((Link<TLink>)other);
72
73 public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
74     && _equalityComparer.Equals(Source, other.Source)
75     && _equalityComparer.Equals(Target, other.Target);
76
77 public static string ToString(TLink index, TLink source, TLink target) => $"{index}:
78     ↳ {source}->{target}";
79
80 public static string ToString(TLink source, TLink target) => $"{source}->{target}";
81
82 public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
83
84 public static implicit operator Link<TLink> (TLink[] linkArray) => new Link<TLink>(linkArray);
85
86 public TLink[] ToArray()
87 {
88     var array = new TLink[Length];
89     CopyTo(array, 0);
90     return array;
91 }
92
93 public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
94     ↳ ToString(Source, Target) : ToString(Index, Source, Target);
95
96 #region IList
97
98 public int Count => Length;
99
100 public bool IsReadOnly => true;
101
102 public TLink this[int index]
103 {
104     get
105     {

```

```

96         Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1), nameof(index));
97         if (index == _constants.IndexPart)
98         {
99             return Index;
100         }
101         if (index == _constants.SourcePart)
102         {
103             return Source;
104         }
105         if (index == _constants.TargetPart)
106         {
107             return Target;
108         }
109         throw new NotSupportedException(); // Impossible path due to Ensure.ArgumentInRange
110     }
111     set => throw new NotSupportedException();
112 }
113
114 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
115
116 public IEnumerator<TLink> GetEnumerator()
117 {
118     yield return Index;
119     yield return Source;
120     yield return Target;
121 }
122
123 public void Add(TLink item) => throw new NotSupportedException();
124
125 public void Clear() => throw new NotSupportedException();
126
127 public bool Contains(TLink item) => IndexOf(item) >= 0;
128
129 public void CopyTo(TLink[] array, int arrayIndex)
130 {
131     Ensure.Always.ArgumentNotNull(array, nameof(array));
132     Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
133         ↪ nameof(arrayIndex));
134     if (arrayIndex + Length > array.Length)
135     {
136         throw new InvalidOperationException();
137     }
138     array[arrayIndex++] = Index;
139     array[arrayIndex++] = Source;
140     array[arrayIndex] = Target;
141 }
142
143 public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
144
145 public int IndexOf(TLink item)
146 {
147     if (_equalityComparer.Equals(Index, item))
148     {
149         return _constants.IndexPart;
150     }
151     if (_equalityComparer.Equals(Source, item))
152     {
153         return _constants.SourcePart;
154     }
155     if (_equalityComparer.Equals(Target, item))
156     {
157         return _constants.TargetPart;
158     }
159     return -1;
160 }
161
162 public void Insert(int index, TLink item) => throw new NotSupportedException();
163
164 public void RemoveAt(int index) => throw new NotSupportedException();
165
166 #endregion
167 }

```

#### ./LinkExtensions.cs

```

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

```

#### ./LinksOperatorBase.cs

```

1 namespace Platform.Data.Doublets
2 {

```

```

3     public abstract class LinksOperatorBase<TLink>
4     {
5         protected readonly ILinks<TLink> Links;
6         protected LinksOperatorBase(ILinks<TLink> links) => Links = links;
7     }
8 }

```

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

```

1  //-----
2  // <auto-generated>
3  //     Generated by the MSBuild WriteCodeFragment class.
4  // </auto-generated>
5  //-----
6
7  using System;
8  using System.Reflection;
9
10 [assembly: System.Reflection.AssemblyConfigurationAttribute("Debug")]
11 [assembly: System.Reflection.AssemblyCopyrightAttribute("Konstantin Diachenko")]
12 [assembly: System.Reflection.AssemblyDescriptionAttribute("LinksPlatform\'s Platform.Data.Doublets
   ↳ Class Library")]
13 [assembly: System.Reflection.AssemblyFileVersionAttribute("0.0.1.0")]
14 [assembly: System.Reflection.AssemblyInformationalVersionAttribute("0.0.1")]
15 [assembly: System.Reflection.AssemblyTitleAttribute("Platform.Data.Doublets")]
16 [assembly: System.Reflection.AssemblyVersionAttribute("0.0.1.0")]

```

#### ./PropertyOperators/DefaultLinkPropertyOperator.cs

```

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

```

#### ./PropertyOperators/FrequencyPropertyOperator.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.PropertyOperators
5  {
6      public class FrequencyPropertyOperator<TLink> : LinksOperatorBase<TLink>,
   ↳ ISpecificPropertyOperator<TLink, TLink>
7      {
8          private static readonly EqualityComparer<TLink> _equalityComparer =
   ↳ EqualityComparer<TLink>.Default;
9
10         private readonly TLink _frequencyPropertyMarker;
11         private readonly TLink _frequencyMarker;
12
13         public FrequencyPropertyOperator(ILinks<TLink> links, TLink frequencyPropertyMarker, TLink
   ↳ frequencyMarker) : base(links)
14         {
15             _frequencyPropertyMarker = frequencyPropertyMarker;

```

```

16         _frequencyMarker = frequencyMarker;
17     }
18
19     public TLink Get(TLink link)
20     {
21         var property = Links.SearchOrDefault(link, _frequencyPropertyMarker);
22         var container = GetContainer(property);
23         var frequency = GetFrequency(container);
24         return frequency;
25     }
26
27     private TLink GetContainer(TLink property)
28     {
29         var frequencyContainer = default(TLink);
30         if (_equalityComparer.Equals(property, default))
31         {
32             return frequencyContainer;
33         }
34         Links.Each(candidate =>
35         {
36             var candidateTarget = Links.GetTarget(candidate);
37             var frequencyTarget = Links.GetTarget(candidateTarget);
38             if (_equalityComparer.Equals(frequencyTarget, _frequencyMarker))
39             {
40                 frequencyContainer = Links.GetIndex(candidate);
41                 return Links.Constants.Break;
42             }
43             return Links.Constants.Continue;
44         }, Links.Constants.Any, property, Links.Constants.Any);
45         return frequencyContainer;
46     }
47
48     private TLink GetFrequency(TLink container) => _equalityComparer.Equals(container, default) ?
49         ↪ default : Links.GetTarget(container);
50
51     public void Set(TLink link, TLink frequency)
52     {
53         var property = Links.GetOrCreate(link, _frequencyPropertyMarker);
54         var container = GetContainer(property);
55         if (_equalityComparer.Equals(container, default))
56         {
57             Links.GetOrCreate(property, frequency);
58         }
59         else
60         {
61             Links.Update(container, property, frequency);
62         }
63     }
64 }

```

#### ./ResizableDirectMemory/ResizableDirectMemoryLinks.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using System.Runtime.InteropServices;
5  using Platform.Disposables;
6  using Platform.Helpers.Singletons;
7  using Platform.Collections.Arrays;
8  using Platform.Numbers;
9  using Platform.Unsafe;
10 using Platform.Memory;
11 using Platform.Data.Exceptions;
12 using Platform.Data.Constants;
13 using static Platform.Numbers.ArithmeticHelpers;
14
15 #pragma warning disable 0649
16 #pragma warning disable 169
17 #pragma warning disable 618
18
19 // ReSharper disable StaticMemberInGenericType
20 // ReSharper disable BuiltInTypeReferenceStyle
21 // ReSharper disable MemberCanBePrivate.Local
22 // ReSharper disable UnusedMember.Local
23
24 namespace Platform.Data.Doublets.ResizableDirectMemory
25 {
26     public partial class ResizableDirectMemoryLinks<TLink> : DisposableBase, ILinks<TLink>
27     {
28         private static readonly EqualityComparer<TLink> _equalityComparer =
29             ↪ EqualityComparer<TLink>.Default;
30         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
31
32         /// <summary>Возвращает размер одной связи в байтах.</summary>
33         public static readonly int LinkSizeInBytes = StructureHelpers.SizeOf<Link>();
34
35         public static readonly int LinkHeaderSizeInBytes = StructureHelpers.SizeOf<LinksHeader>();
36
37         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;

```

```

37
38 private struct Link
39 {
40     public static readonly int SourceOffset = Marshal.OffsetOf(typeof(Link),
41         ↳ nameof(Source)).ToInt32();
42     public static readonly int TargetOffset = Marshal.OffsetOf(typeof(Link),
43         ↳ nameof(Target)).ToInt32();
44     public static readonly int LeftAsSourceOffset = Marshal.OffsetOf(typeof(Link),
45         ↳ nameof(LeftAsSource)).ToInt32();
46     public static readonly int RightAsSourceOffset = Marshal.OffsetOf(typeof(Link),
47         ↳ nameof(RightAsSource)).ToInt32();
48     public static readonly int SizeAsSourceOffset = Marshal.OffsetOf(typeof(Link),
49         ↳ nameof(SizeAsSource)).ToInt32();
50     public static readonly int LeftAsTargetOffset = Marshal.OffsetOf(typeof(Link),
51         ↳ nameof(LeftAsTarget)).ToInt32();
52     public static readonly int RightAsTargetOffset = Marshal.OffsetOf(typeof(Link),
53         ↳ nameof(RightAsTarget)).ToInt32();
54     public static readonly int SizeAsTargetOffset = Marshal.OffsetOf(typeof(Link),
55         ↳ nameof(SizeAsTarget)).ToInt32();
56
57     public TLink Source;
58     public TLink Target;
59     public TLink LeftAsSource;
60     public TLink RightAsSource;
61     public TLink SizeAsSource;
62     public TLink LeftAsTarget;
63     public TLink RightAsTarget;
64     public TLink SizeAsTarget;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     public static TLink GetSource(IntPtr pointer) => (pointer + SourceOffset).GetValue<TLink>();
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     public static TLink GetTarget(IntPtr pointer) => (pointer + TargetOffset).GetValue<TLink>();
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     public static TLink GetLeftAsSource(IntPtr pointer) => (pointer +
72         ↳ LeftAsSourceOffset).GetValue<TLink>();
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public static TLink GetRightAsSource(IntPtr pointer) => (pointer +
75         ↳ RightAsSourceOffset).GetValue<TLink>();
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     public static TLink GetSizeAsSource(IntPtr pointer) => (pointer +
78         ↳ SizeAsSourceOffset).GetValue<TLink>();
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     public static TLink GetLeftAsTarget(IntPtr pointer) => (pointer +
81         ↳ LeftAsTargetOffset).GetValue<TLink>();
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     public static TLink GetRightAsTarget(IntPtr pointer) => (pointer +
84         ↳ RightAsTargetOffset).GetValue<TLink>();
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public static TLink GetSizeAsTarget(IntPtr pointer) => (pointer +
87         ↳ SizeAsTargetOffset).GetValue<TLink>();
88
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     public static void SetSource(IntPtr pointer, TLink value) => (pointer +
91         ↳ SourceOffset).SetValue(value);
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public static void SetTarget(IntPtr pointer, TLink value) => (pointer +
94         ↳ TargetOffset).SetValue(value);
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     public static void SetLeftAsSource(IntPtr pointer, TLink value) => (pointer +
97         ↳ LeftAsSourceOffset).SetValue(value);
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     public static void SetRightAsSource(IntPtr pointer, TLink value) => (pointer +
100         ↳ RightAsSourceOffset).SetValue(value);
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     public static void SetSizeAsSource(IntPtr pointer, TLink value) => (pointer +
103         ↳ SizeAsSourceOffset).SetValue(value);
104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
105     public static void SetLeftAsTarget(IntPtr pointer, TLink value) => (pointer +
106         ↳ LeftAsTargetOffset).SetValue(value);
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     public static void SetRightAsTarget(IntPtr pointer, TLink value) => (pointer +
109         ↳ RightAsTargetOffset).SetValue(value);
110     [MethodImpl(MethodImplOptions.AggressiveInlining)]
111     public static void SetSizeAsTarget(IntPtr pointer, TLink value) => (pointer +
112         ↳ SizeAsTargetOffset).SetValue(value);
113 }
114
115 private struct LinksHeader
116 {
117     public static readonly int AllocatedLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
118         ↳ nameof(AllocatedLinks)).ToInt32();
119     public static readonly int ReservedLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
120         ↳ nameof(ReservedLinks)).ToInt32();

```

```

97     public static readonly int FreeLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
98         ↳ nameof(FreeLinks)).ToInt32();
99     public static readonly int FirstFreeLinkOffset = Marshal.OffsetOf(typeof(LinksHeader),
100        ↳ nameof(FirstFreeLink)).ToInt32();
101     public static readonly int FirstAsSourceOffset = Marshal.OffsetOf(typeof(LinksHeader),
102        ↳ nameof(FirstAsSource)).ToInt32();
103     public static readonly int FirstAsTargetOffset = Marshal.OffsetOf(typeof(LinksHeader),
104        ↳ nameof(FirstAsTarget)).ToInt32();
105     public static readonly int LastFreeLinkOffset = Marshal.OffsetOf(typeof(LinksHeader),
106        ↳ nameof(LastFreeLink)).ToInt32();
107
108     public TLink AllocatedLinks;
109     public TLink ReservedLinks;
110     public TLink FreeLinks;
111     public TLink FirstFreeLink;
112     public TLink FirstAsSource;
113     public TLink FirstAsTarget;
114     public TLink LastFreeLink;
115     public TLink Reserved8;
116
117     [MethodImpl(MethodImplOptions.AggressiveInlining)]
118     public static TLink GetAllocatedLinks(IntPtr pointer) => (pointer +
119         ↳ AllocatedLinksOffset).GetValue<TLink>();
120     [MethodImpl(MethodImplOptions.AggressiveInlining)]
121     public static TLink GetReservedLinks(IntPtr pointer) => (pointer +
122         ↳ ReservedLinksOffset).GetValue<TLink>();
123     [MethodImpl(MethodImplOptions.AggressiveInlining)]
124     public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
125         ↳ FreeLinksOffset).GetValue<TLink>();
126     [MethodImpl(MethodImplOptions.AggressiveInlining)]
127     public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
128         ↳ FirstFreeLinkOffset).GetValue<TLink>();
129     [MethodImpl(MethodImplOptions.AggressiveInlining)]
130     public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
131         ↳ FirstAsSourceOffset).GetValue<TLink>();
132     [MethodImpl(MethodImplOptions.AggressiveInlining)]
133     public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
134         ↳ FirstAsTargetOffset).GetValue<TLink>();
135     [MethodImpl(MethodImplOptions.AggressiveInlining)]
136     public static TLink GetLastFreeLink(IntPtr pointer) => (pointer +
137         ↳ LastFreeLinkOffset).GetValue<TLink>();
138
139     [MethodImpl(MethodImplOptions.AggressiveInlining)]
140     public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) => pointer +
141         ↳ FirstAsSourceOffset;
142     [MethodImpl(MethodImplOptions.AggressiveInlining)]
143     public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
144         ↳ FirstAsTargetOffset;
145
146     [MethodImpl(MethodImplOptions.AggressiveInlining)]
147     public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
148         ↳ AllocatedLinksOffset).SetValue(value);
149     [MethodImpl(MethodImplOptions.AggressiveInlining)]
150     public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
151         ↳ ReservedLinksOffset).SetValue(value);
152     [MethodImpl(MethodImplOptions.AggressiveInlining)]
153     public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
154         ↳ FreeLinksOffset).SetValue(value);
155     [MethodImpl(MethodImplOptions.AggressiveInlining)]
156     public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
157         ↳ FirstFreeLinkOffset).SetValue(value);
158     [MethodImpl(MethodImplOptions.AggressiveInlining)]
159     public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
160         ↳ FirstAsSourceOffset).SetValue(value);
161     [MethodImpl(MethodImplOptions.AggressiveInlining)]
162     public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
163         ↳ FirstAsTargetOffset).SetValue(value);
164     [MethodImpl(MethodImplOptions.AggressiveInlining)]
165     public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
166         ↳ LastFreeLinkOffset).SetValue(value);
167 }
168
169 private readonly long _memoryReservationStep;
170
171 private readonly IResizableDirectMemory _memory;
172 private IntPtr _header;
173 private IntPtr _links;
174
175 private LinksTargetsTreeMethods _targetsTreeMethods;
176 private LinksSourcesTreeMethods _sourcesTreeMethods;
177
178 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой, нужно
179 ↳ использовать не список а дерево, так как так можно быстрее проверить на наличие связи внутри
180 private UnusedLinksListMethods _unusedLinksListMethods;
181
182 /// <summary>

```

```

161     /// Возвращает общее число связей находящихся в хранилище.
162     /// </summary>
163     private TLink Total => Subtract(LinksHeader.GetAllocatedLinks(_header),
164     ↪ LinksHeader.GetFreeLinks(_header));
165
166     public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
167
168     public ResizableDirectMemoryLinks(string address)
169     : this(address, DefaultLinksSizeStep)
170     {
171     }
172
173     /// <summary>
174     /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным минимальным
175     ↪ шагом расширения базы данных.
176     /// </summary>
177     /// <param name="address">Полный путь к файлу базы данных.</param>
178     /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в байтах.</param>
179     public ResizableDirectMemoryLinks(string address, long memoryReservationStep)
180     : this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
181     ↪ memoryReservationStep)
182     {
183     }
184
185     public ResizableDirectMemoryLinks(IResizableDirectMemory memory)
186     : this(memory, DefaultLinksSizeStep)
187     {
188     }
189
190     public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long 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             }
249         }
250     }

```

```

242         var storedLinkValue = GetLinkUnsafe(index);
243         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
244             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
245         {
246             return Integer<TLink>.One;
247         }
248         return Integer<TLink>.Zero;
249     }
250 }
251 if (restrictions.Count == 3)
252 {
253     var index = restrictions[Constants.IndexPart];
254     var source = restrictions[Constants.SourcePart];
255     var target = restrictions[Constants.TargetPart];
256
257     if (_equalityComparer.Equals(index, Constants.Any))
258     {
259         if (_equalityComparer.Equals(source, Constants.Any) &&
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) ? Integer<TLink>.Zero :
277                 ↪ 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         var value = default(TLink);
303         if (_equalityComparer.Equals(source, Constants.Any))
304         {
305             value = target;
306         }
307         if (_equalityComparer.Equals(target, Constants.Any))
308         {
309             value = source;
310         }
311         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
312             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
313         {
314             return Integer<TLink>.One;
315         }
316         return Integer<TLink>.Zero;
317     }
318 }
319 throw new NotSupportedException("Другие размеры и способы ограничений не поддерживаются.");
320 }
321
322 [MethodImpl(MethodImplOptions.AggressiveInlining)]
323 public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
324 {
325     if (restrictions.Count == 0)
326     {

```



```

323     for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
324         ↳ (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
325         ↳ Increment(link))
326     {
327         if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
328             ↳ Constants.Break))
329         {
330             return Constants.Break;
331         }
332     }
333     return Constants.Continue;
334 }
335 if (restrictions.Count == 1)
336 {
337     var index = restrictions[Constants.IndexPart];
338     if (_equalityComparer.Equals(index, Constants.Any))
339     {
340         return Each(handler, ArrayPool<TLink>.Empty);
341     }
342     if (!Exists(index))
343     {
344         return Constants.Continue;
345     }
346     return handler(GetLinkStruct(index));
347 }
348 if (restrictions.Count == 2)
349 {
350     var index = restrictions[Constants.IndexPart];
351     var value = restrictions[1];
352     if (_equalityComparer.Equals(index, Constants.Any))
353     {
354         if (_equalityComparer.Equals(value, Constants.Any))
355         {
356             return Each(handler, ArrayPool<TLink>.Empty);
357         }
358         if (_equalityComparer.Equals(Each(handler, new[] { index, value, Constants.Any }),
359             ↳ Constants.Break))
360         {
361             return Constants.Break;
362         }
363         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         {
398             return _targetsTreeMethods.EachReference(target, handler);
399         }
400         else if (_equalityComparer.Equals(target, Constants.Any))
401         {
402             return _sourcesTreeMethods.EachReference(source, handler);
403         }
404         else //if(source != Any && target != Any)
405         {
406             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) &&
420             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
421         {
422             return handler(GetLinkStruct(index));
423         }
424         return Constants.Continue;
425     }
426     var value = default(TLink);
427     if (_equalityComparer.Equals(source, Constants.Any))
428     {
429         value = target;
430     }
431     if (_equalityComparer.Equals(target, Constants.Any))
432     {
433         value = source;
434     }
435     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
436         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
437     {
438         return handler(GetLinkStruct(index));
439     }
440     return Constants.Continue;
441 }
442 }
443 throw new NotSupportedException("Другие размеры и способы ограничений не поддерживаются.");
444 }
445
446 /// <remarks>
447 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует в
           ↪ другом месте (но не в менеджере памяти, а в логике Links)
448 /// </remarks>
449 [MethodImpl(MethodImplOptions.AggressiveInlining)]
450 public TLink Update(IList<TLink> values)
451 {
452     var linkIndex = values[Constants.IndexPart];
453     var link = GetLinkUnsafe(linkIndex);
454     // Будет корректно работать только в том случае, если пространство выделенной связи
           ↪ предварительно заполнено нулями
455     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
456     {
457         _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header), linkIndex);
458     }
459     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
460     {
461         _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header), linkIndex);
462     }
463     Link.SetSource(link, values[Constants.SourcePart]);
464     Link.SetTarget(link, values[Constants.TargetPart]);
465     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
466     {
467         _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer(_header), linkIndex);
468     }
469     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
470     {
471         _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTargetPointer(_header), linkIndex);
472     }
473     return linkIndex;
474 }
475
476 [MethodImpl(MethodImplOptions.AggressiveInlining)]
477 public Link<TLink> GetLinkStruct(TLink linkIndex)
478 {
479     var link = GetLinkUnsafe(linkIndex);
480     return new Link<TLink>(linkIndex, Link.GetSource(link), Link.GetTarget(link));
481 }
482
483 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

484 private IntPtr GetLinkUnsafe(TLink linkIndex) => _links.GetElement(LinkSizeInBytes, linkIndex);
485
486 /// <remarks>
487 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет пространство
488 /// </remarks>
489 public TLink Create()
490 {
491     var freeLink = LinksHeader.GetFirstFreeLink(_header);
492     if (!_equalityComparer.Equals(freeLink, Constants.Null))
493     {
494         _unusedLinksListMethods.Detach(freeLink);
495     }
496     else
497     {
498         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
499             ↳ Constants.MaxPossibleIndex) > 0)
500         {
501             throw new LinksLimitReachedException((Integer<TLink>)Constants.MaxPossibleIndex);
502         }
503         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
504             ↳ Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
505         {
506             _memory.ReservedCapacity += _memoryReservationStep;
507             SetPointers(_memory);
508             LinksHeader.SetReservedLinks(_header, (Integer<TLink>)(_memory.ReservedCapacity /
509                 ↳ LinkSizeInBytes));
510         }
511         LinksHeader.SetAllocatedLinks(_header,
512             ↳ Increment(LinksHeader.GetAllocatedLinks(_header)));
513         _memory.UsedCapacity += LinkSizeInBytes;
514         freeLink = LinksHeader.GetAllocatedLinks(_header);
515     }
516     return freeLink;
517 }
518
519 public void Delete(TLink link)
520 {
521     if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)
522     {
523         _unusedLinksListMethods.AttachAsFirst(link);
524     }
525     else if (_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
526     {
527         LinksHeader.SetAllocatedLinks(_header,
528             ↳ Decrement(LinksHeader.GetAllocatedLinks(_header)));
529         _memory.UsedCapacity -= LinkSizeInBytes;
530         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор, пока не
531         //   дойдём до первой существующей связи
532         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
533         while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header), Integer<TLink>.Zero)
534             ↳ > 0) && IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
535         {
536             _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
537             LinksHeader.SetAllocatedLinks(_header,
538                 ↳ Decrement(LinksHeader.GetAllocatedLinks(_header)));
539             _memory.UsedCapacity -= LinkSizeInBytes;
540         }
541     }
542 }
543
544 /// <remarks>
545 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если адрес
546   ↳ реально поменялся
547 ///
548 /// Указатель this.links может быть в том же месте,
549 /// так как 0-я связь не используется и имеет такой же размер как Header,
550 /// поэтому header размещается в том же месте, что и 0-я связь
551 /// </remarks>
552 private void SetPointers(IDirectMemory memory)
553 {
554     if (memory == null)
555     {
556         _links = IntPtr.Zero;
557         _header = _links;
558         _unusedLinksListMethods = null;
559         _targetsTreeMethods = null;
560         _unusedLinksListMethods = null;
561     }
562     else
563     {
564         _links = memory.Pointer;
565         _header = _links;
566         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
567         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
568         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
569     }
570 }
571

```

```

562 [MethodImpl(MethodImplOptions.AggressiveInlining)]
563 private bool Exists(TLink link)
564     => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
565     && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
566     && !IsUnusedLink(link);
567
568 [MethodImpl(MethodImplOptions.AggressiveInlining)]
569 private bool IsUnusedLink(TLink link)
570     => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
571     || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)), Constants.Null)
572     && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)), Constants.Null));
573
574 #region DisposableBase
575
576 protected override bool AllowMultipleDisposeCalls => true;
577
578 protected override void DisposeCore(bool manual, bool wasDisposed)
579 {
580     if (!wasDisposed)
581     {
582         SetPointers(null);
583     }
584     Disposable.TryDispose(_memory);
585 }
586
587 #endregion
588 }
589 }
590

```

#### ./ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs

```

1  using System;
2  using Platform.Unsafe;
3  using Platform.Collections.Methods.Lists;
4
5  namespace Platform.Data.Doublets.ResizableDirectMemory
6  {
7      partial class ResizableDirectMemoryLinks<TLink>
8      {
9          private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<TLink>
10          {
11              private readonly IntPtr _links;
12              private readonly IntPtr _header;
13
14              public UnusedLinksListMethods(IntPtr links, IntPtr header)
15              {
16                  _links = links;
17                  _header = header;
18              }
19
20              protected override TLink GetFirst() => (_header +
21                  ↳ LinksHeader.FirstFreeLinkOffset).GetValue<TLink>();
22
23              protected override TLink GetLast() => (_header +
24                  ↳ LinksHeader.LastFreeLinkOffset).GetValue<TLink>();
25
26              protected override TLink GetPrevious(TLink element) => (_links.GetElement(LinkSizeInBytes,
27                  ↳ element) + Link.SourceOffset).GetValue<TLink>();
28
29              protected override TLink GetNext(TLink element) => (_links.GetElement(LinkSizeInBytes,
30                  ↳ element) + Link.TargetOffset).GetValue<TLink>();
31
32              protected override TLink GetSize() => (_header +
33                  ↳ LinksHeader.FreeLinksOffset).GetValue<TLink>();
34
35              protected override void SetFirst(TLink element) => (_header +
36                  ↳ LinksHeader.FirstFreeLinkOffset).SetValue(element);
37
38              protected override void SetLast(TLink element) => (_header +
39                  ↳ LinksHeader.LastFreeLinkOffset).SetValue(element);
40
41              protected override void SetPrevious(TLink element, TLink previous) =>
42                  ↳ (_links.GetElement(LinkSizeInBytes, element) + Link.SourceOffset).SetValue(previous);
43
44              protected override void SetNext(TLink element, TLink next) =>
45                  ↳ (_links.GetElement(LinkSizeInBytes, element) + Link.TargetOffset).SetValue(next);
46
47              protected override void SetSize(TLink size) => (_header +
48                  ↳ LinksHeader.FreeLinksOffset).SetValue(size);
49          }
50      }
51

```

#### ./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs

```

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

```

```

5  using Platform.Numbers;
6  using Platform.Unsafe;
7  using Platform.Collections.Methods.Trees;
8  using Platform.Data.Constants;
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory
11 {
12     partial class ResizableDirectMemoryLinks<TLink>
13     {
14         private abstract class LinksTreeMethodsBase : SizedAndThreadedAVLBalancedTreeMethods<TLink>
15         {
16             private readonly ResizableDirectMemoryLinks<TLink> _memory;
17             private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
18             protected readonly IntPtr Links;
19             protected readonly IntPtr Header;
20
21             protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
22             {
23                 Links = memory._links;
24                 Header = memory._header;
25                 _memory = memory;
26                 _constants = memory.Constants;
27             }
28
29             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             protected abstract TLink GetTreeRoot();
31
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             protected abstract TLink GetBasePartValue(TLink link);
34
35             public TLink this[TLink index]
36             {
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
61                                     ↳ broken)
62                 }
63             }
64
65             // TODO: Return indices range instead of references count
66             public TLink CalculateReferences(TLink link)
67             {
68                 var root = GetTreeRoot();
69                 var total = GetSize(root);
70                 var totalRightIgnore = GetZero();
71                 while (!EqualToZero(root))
72                 {
73                     var @base = GetBasePartValue(root);
74                     if (LessOrEqualThan(@base, link))
75                     {
76                         root = GetRightOrDefault(root);
77                     }
78                     else
79                     {
80                         totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
81                         root = GetLeftOrDefault(root);
82                     }
83                 }
84                 root = GetTreeRoot();
85                 var totalLeftIgnore = GetZero();
86                 while (!EqualToZero(root))
87                 {
88                     var @base = GetBasePartValue(root);
89                     if (GreaterOrEqualThan(@base, link))
90                     {
91                         root = GetLeftOrDefault(root);
92                     }
93                 }
94             }
95         }
96     }
97 }

```

```

92         else
93         {
94             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
95
96             root = GetRightOrDefault(root);
97         }
98     }
99     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
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) => Links.GetElement(LinkSizeInBytes,
164         ↪ node) + Link.LeftAsSourceOffset;
165
166     protected override IntPtr GetRightPointer(TLink node) => Links.GetElement(LinkSizeInBytes,
167         ↪ node) + Link.RightAsSourceOffset;
168
169     protected override TLink GetLeftValue(TLink node) => (Links.GetElement(LinkSizeInBytes,
170         ↪ node) + Link.LeftAsSourceOffset).GetValue<TLink>();
171
172     protected override TLink GetRightValue(TLink node) => (Links.GetElement(LinkSizeInBytes,
173         ↪ node) + Link.RightAsSourceOffset).GetValue<TLink>();
174
175     protected override TLink GetSize(TLink node)
176     {
177     }
178 }

```

```

172     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
173         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
174     return BitwiseHelpers.PartialRead(previousValue, 5, -5);
175 }
176
177 protected override void SetLeft(TLink node, TLink left) =>
178     ↪ (Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset).SetValue(left);
179
180 protected override void SetRight(TLink node, TLink right) =>
181     ↪ (Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset).SetValue(right);
182
183 protected override void SetSize(TLink node, TLink size)
184 {
185     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
186         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
187     (Links.GetElement(LinkSizeInBytes, node) +
188         ↪ Link.SizeAsSourceOffset).SetValue(BitwiseHelpers.PartialWrite(previousValue, size,
189         ↪ 5, -5));
190 }
191
192 protected override bool GetLeftIsChild(TLink node)
193 {
194     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
195         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
196     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 4, 1);
197 }
198
199 protected override void SetLeftIsChild(TLink node, bool value)
200 {
201     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
202         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
203     var modified = BitwiseHelpers.PartialWrite(previousValue, (TLink)(Integer<TLink>)value,
204         ↪ 4, 1);
205     (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsSourceOffset).SetValue(modified);
206 }
207
208 protected override bool GetRightIsChild(TLink node)
209 {
210     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
211         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
212     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 3, 1);
213 }
214
215 protected override void SetRightIsChild(TLink node, bool value)
216 {
217     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
218         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
219     var modified = BitwiseHelpers.PartialWrite(previousValue, (TLink)(Integer<TLink>)value,
220         ↪ 3, 1);
221     (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsSourceOffset).SetValue(modified);
222 }
223
224 protected override sbyte GetBalance(TLink node)
225 {
226     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
227         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
228     var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 0, 3);
229     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 | 124 :
230         ↪ 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 & 3;
239     var modified = BitwiseHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
240     (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsSourceOffset).SetValue(modified);
241 }
242
243 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
244 {
245     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
246         ↪ Link.SourceOffset).GetValue<TLink>();
247     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
248         ↪ Link.SourceOffset).GetValue<TLink>();
249     return LessThan(firstSource, secondSource) ||
250         (Equals(firstSource, secondSource) &&
251         ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
252         ↪ Link.TargetOffset).GetValue<TLink>(), (Links.GetElement(LinkSizeInBytes,
253         ↪ second) + Link.TargetOffset).GetValue<TLink>()));
254 }
255
256 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) => Links.GetElement(LinkSizeInBytes,
    ↪ node) + Link.LeftAsTargetOffset;
293
294     protected override IntPtr GetRightPointer(TLink node) => Links.GetElement(LinkSizeInBytes,
    ↪ node) + Link.RightAsTargetOffset;
295
296     protected override TLink GetLeftValue(TLink node) => (Links.GetElement(LinkSizeInBytes,
    ↪ node) + Link.LeftAsTargetOffset).GetValue<TLink>();
297
298     protected override TLink GetRightValue(TLink node) => (Links.GetElement(LinkSizeInBytes,
    ↪ node) + Link.RightAsTargetOffset).GetValue<TLink>();
299
300     protected override TLink GetSize(TLink node)
301     {

```



```

302     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
303         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
304     return BitwiseHelpers.PartialRead(previousValue, 5, -5);
305 }
306
307 protected override void SetLeft(TLink node, TLink left) =>
308     ↪ (Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset).SetValue(left);
309
310 protected override void SetRight(TLink node, TLink right) =>
311     ↪ (Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset).SetValue(right);
312
313 protected override void SetSize(TLink node, TLink size)
314 {
315     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
316         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
317     (Links.GetElement(LinkSizeInBytes, node) +
318         ↪ Link.SizeAsTargetOffset).SetValue(BitwiseHelpers.PartialWrite(previousValue, size,
319         ↪ 5, -5));
320 }
321
322 protected override bool GetLeftIsChild(TLink node)
323 {
324     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
325         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
326     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 4, 1);
327 }
328
329 protected override void SetLeftIsChild(TLink node, bool value)
330 {
331     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
332         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
333     var modified = BitwiseHelpers.PartialWrite(previousValue, (TLink)(Integer<TLink>)value,
334         ↪ 4, 1);
335     (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsTargetOffset).SetValue(modified);
336 }
337
338 protected override bool GetRightIsChild(TLink node)
339 {
340     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
341         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
342     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 3, 1);
343 }
344
345 protected override void SetRightIsChild(TLink node, bool value)
346 {
347     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
348         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
349     var modified = BitwiseHelpers.PartialWrite(previousValue, (TLink)(Integer<TLink>)value,
350         ↪ 3, 1);
351     (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsTargetOffset).SetValue(modified);
352 }
353
354 protected override sbyte GetBalance(TLink node)
355 {
356     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
357         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
358     var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 0, 3);
359     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 | 124 :
360         ↪ value & 3);
361     return unpackedValue;
362 }
363
364 protected override void SetBalance(TLink node, sbyte value)
365 {
366     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
367         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
368     var packagedValue = (TLink)(Integer<TLink>)(((byte)value >> 5) & 4) | value & 3;
369     var modified = BitwiseHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
370     (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsTargetOffset).SetValue(modified);
371 }
372
373 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
374 {
375     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
376         ↪ Link.TargetOffset).GetValue<TLink>();
377     var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
378         ↪ Link.TargetOffset).GetValue<TLink>();
379     return LessThan(firstTarget, secondTarget) ||
380         (IsEquals(firstTarget, secondTarget) &&
381         ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
382         ↪ Link.SourceOffset).GetValue<TLink>(), (Links.GetElement(LinkSizeInBytes,
383         ↪ second) + Link.SourceOffset).GetValue<TLink>()));
384 }
385
386 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)

```

```

367     {
368         var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
369             ↪ Link.TargetOffset).GetValue<TLink>();
370         var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
371             ↪ Link.TargetOffset).GetValue<TLink>();
372         return GreaterThan(firstTarget, secondTarget) ||
373             (IsEquals(firstTarget, secondTarget) &&
374                 ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
375                 ↪ Link.SourceOffset).GetValue<TLink>(), (Links.GetElement(LinkSizeInBytes,
376                 ↪ second) + Link.SourceOffset).GetValue<TLink>()));
377     }
378 }
379 }

```

## ./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Collections.Arrays;
6  using Platform.Helpers.Singletons;
7  using Platform.Memory;
8  using Platform.Data.Exceptions;
9  using Platform.Data.Constants;
10
11  //#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
12
13  #pragma warning disable 0649
14  #pragma warning disable 169
15
16  // ReSharper disable BuiltInTypeReferenceStyle
17
18  namespace Platform.Data.Doublets.ResizableDirectMemory
19  {
20      using id = UInt64;
21
22      public unsafe partial class UInt64ResizableDirectMemoryLinks : DisposableBase, ILinks<id>
23      {
24          /// <summary>Возвращает размер одной связи в байтах.</summary>
25          /// <remarks>
26          ///     Используется только во вне класса, не рекомендуется использовать внутри.
27          ///     Так как во вне не обязательно будет доступен unsafe C#.
28          /// </remarks>
29          public static readonly int LinkSizeInBytes = sizeof(Link);
30
31          public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
32
33          private struct Link
34          {
35              public id Source;
36              public id Target;
37              public id LeftAsSource;
38              public id RightAsSource;
39              public id SizeAsSource;
40              public id LeftAsTarget;
41              public id RightAsTarget;
42              public id SizeAsTarget;
43          }
44
45          private struct LinksHeader
46          {
47              public id AllocatedLinks;
48              public id ReservedLinks;
49              public id FreeLinks;
50              public id FirstFreeLink;
51              public id FirstAsSource;
52              public id FirstAsTarget;
53              public id LastFreeLink;
54              public id Reserved8;
55          }
56
57          private readonly long _memoryReservationStep;
58
59          private readonly IResizableDirectMemory _memory;
60          private LinksHeader* _header;
61          private Link* _links;
62
63          private LinksTargetsTreeMethods _targetsTreeMethods;
64          private LinksSourcesTreeMethods _sourcesTreeMethods;
65
66          // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой, нужно
67          ↪ использовать не список а дерево, так как так можно быстрее проверить на наличие связи внутри
68          private UnusedLinksListMethods _unusedLinksListMethods;

```

```

68
69 /// <summary>
70 /// Возвращает общее число связей находящихся в хранилище.
71 /// </summary>
72 private id Total => _header->AllocatedLinks - _header->FreeLinks;
73
74 // TODO: Дать возможность переопределять в конструкторе
75 public LinksCombinedConstants<id, id, int> Constants { get; }
76
77 public UInt64ResizableDirectMemoryLinks(string address) : this(address, DefaultLinksSizeStep) {
78     ↪ }
79
80 /// <summary>
81 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным минимальным
82     ↪ шагом расширения базы данных.
83 /// </summary>
84 /// <param name="address">Полный путь к файлу базы данных.</param>
85 /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в байтах.</param>
86 public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) : this(new
87     ↪ FileMappedResizableDirectMemory(address, memoryReservationStep), memoryReservationStep) { }
88
89 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
90     ↪ DefaultLinksSizeStep) { }
91
92 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
93     ↪ memoryReservationStep)
94 {
95     Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
96     _memory = memory;
97     _memoryReservationStep = memoryReservationStep;
98     if (memory.ReservedCapacity < memoryReservationStep)
99     {
100         memory.ReservedCapacity = memoryReservationStep;
101     }
102     SetPointers(_memory);
103     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
104     _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) + sizeof(LinksHeader);
105     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
106     _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(LinksHeader)) /
107     ↪ sizeof(Link));
108 }
109
110 [MethodImpl(MethodImplOptions.AggressiveInlining)]
111 public id Count(IList<id> restrictions)
112 {
113     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
114     if (restrictions.Count == 0)
115     {
116         return Total;
117     }
118     if (restrictions.Count == 1)
119     {
120         var index = restrictions[Constants.IndexPart];
121         if (index == Constants.Any)
122         {
123             return Total;
124         }
125         return Exists(index) ? 1UL : 0UL;
126     }
127     if (restrictions.Count == 2)
128     {
129         var index = restrictions[Constants.IndexPart];
130         var value = restrictions[1];
131         if (index == Constants.Any)
132         {
133             if (value == Constants.Any)
134             {
135                 return Total; // Any - как отсутствие ограничения
136             }
137             return _sourcesTreeMethods.CalculateReferences(value)
138                 + _targetsTreeMethods.CalculateReferences(value);
139         }
140         else
141         {
142             if (!Exists(index))
143             {
144                 return 0;
145             }
146             if (value == Constants.Any)
147             {
148                 return 1;
149             }
150             var storedLinkValue = GetLinkUnsafe(index);
151             if (storedLinkValue->Source == value ||
152                 storedLinkValue->Target == value)
153             {
154                 return 1;
155             }
156         }
157     }
158 }

```

```

149     }
150     return 0;
151 }
152
153 if (restrictions.Count == 3)
154 {
155     var index = restrictions[Constants.IndexPart];
156     var source = restrictions[Constants.SourcePart];
157     var target = restrictions[Constants.TargetPart];
158     if (index == Constants.Any)
159     {
160         if (source == Constants.Any && target == Constants.Any)
161         {
162             return Total;
163         }
164         else if (source == Constants.Any)
165         {
166             return _targetsTreeMethods.CalculateReferences(target);
167         }
168         else if (target == Constants.Any)
169         {
170             return _sourcesTreeMethods.CalculateReferences(source);
171         }
172         else //if(source != Any && target != Any)
173         {
174             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
175             var link = _sourcesTreeMethods.Search(source, target);
176             return link == Constants.Null ? 0UL : 1UL;
177         }
178     }
179     else
180     {
181         if (!Exists(index))
182         {
183             return 0;
184         }
185         if (source == Constants.Any && target == Constants.Any)
186         {
187             return 1;
188         }
189         var storedLinkValue = GetLinkUnsafe(index);
190         if (source != Constants.Any && target != Constants.Any)
191         {
192             if (storedLinkValue->Source == source &&
193                 storedLinkValue->Target == target)
194             {
195                 return 1;
196             }
197             return 0;
198         }
199         var value = default(id);
200         if (source == Constants.Any)
201         {
202             value = target;
203         }
204         if (target == Constants.Any)
205         {
206             value = source;
207         }
208         if (storedLinkValue->Source == value ||
209             storedLinkValue->Target == value)
210         {
211             return 1;
212         }
213         return 0;
214     }
215 }
216 throw new NotSupportedException("Другие размеры и способы ограничений не поддерживаются.");
217 }
218
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
221 {
222     if (restrictions.Count == 0)
223     {
224         for (id link = 1; link <= _header->AllocatedLinks; link++)
225         {
226             if (Exists(link))
227             {
228                 if (handler(GetLinkStruct(link)) == Constants.Break)
229                 {
230                     return Constants.Break;
231                 }
232             }
233         }
234         return Constants.Continue;
235     }

```

```

236 if (restrictions.Count == 1)
237 {
238     var index = restrictions[Constants.IndexPart];
239     if (index == Constants.Any)
240     {
241         return Each(handler, ArrayPool<ulong>.Empty);
242     }
243     if (!Exists(index))
244     {
245         return Constants.Continue;
246     }
247     return handler(GetLinkStruct(index));
248 }
249 if (restrictions.Count == 2)
250 {
251     var index = restrictions[Constants.IndexPart];
252     var value = restrictions[1];
253     if (index == Constants.Any)
254     {
255         if (value == Constants.Any)
256         {
257             return Each(handler, ArrayPool<ulong>.Empty);
258         }
259         if (Each(handler, new[] { index, value, Constants.Any }) == Constants.Break)
260         {
261             return Constants.Break;
262         }
263         return Each(handler, new[] { index, Constants.Any, value });
264     }
265     else
266     {
267         if (!Exists(index))
268         {
269             return Constants.Continue;
270         }
271         if (value == Constants.Any)
272         {
273             return handler(GetLinkStruct(index));
274         }
275         var storedLinkValue = GetLinkUnsafe(index);
276         if (storedLinkValue->Source == value ||
277             storedLinkValue->Target == value)
278         {
279             return handler(GetLinkStruct(index));
280         }
281         return Constants.Continue;
282     }
283 }
284 if (restrictions.Count == 3)
285 {
286     var index = restrictions[Constants.IndexPart];
287     var source = restrictions[Constants.SourcePart];
288     var target = restrictions[Constants.TargetPart];
289     if (index == Constants.Any)
290     {
291         if (source == Constants.Any && target == Constants.Any)
292         {
293             return Each(handler, ArrayPool<ulong>.Empty);
294         }
295         else if (source == Constants.Any)
296         {
297             return _targetsTreeMethods.EachReference(target, handler);
298         }
299         else if (target == Constants.Any)
300         {
301             return _sourcesTreeMethods.EachReference(source, handler);
302         }
303         else //if(source != Any && target != Any)
304         {
305             var link = _sourcesTreeMethods.Search(source, target);
306             return link == Constants.Null ? Constants.Continue :
307                 ↪ handler(GetLinkStruct(link));
308         }
309     }
310     else
311     {
312         if (!Exists(index))
313         {
314             return Constants.Continue;
315         }
316         if (source == Constants.Any && target == Constants.Any)
317         {
318             return handler(GetLinkStruct(index));
319         }
320         var storedLinkValue = GetLinkUnsafe(index);
321         if (source != Constants.Any && target != Constants.Any)

```

```

321     {
322         if (storedLinkValue->Source == source &&
323             storedLinkValue->Target == target)
324         {
325             return handler(GetLinkStruct(index));
326         }
327         return Constants.Continue;
328     }
329     var value = default(id);
330     if (source == Constants.Any)
331     {
332         value = target;
333     }
334     if (target == Constants.Any)
335     {
336         value = source;
337     }
338     if (storedLinkValue->Source == value ||
339         storedLinkValue->Target == value)
340     {
341         return handler(GetLinkStruct(index));
342     }
343     return Constants.Continue;
344 }
345 }
346 throw new NotSupportedException("Другие размеры и способы ограничений не поддерживаются.");
347 }
348
349 /// <remarks>
350 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует в
351 /// ↪ другом месте (но не в менеджере памяти, а в логике Links)
352 /// </remarks>
353 [MethodImpl(MethodImplOptions.AggressiveInlining)]
354 public id Update(IList<id> values)
355 {
356     var linkIndex = values[Constants.IndexPart];
357     var link = GetLinkUnsafe(linkIndex);
358     // Будет корректно работать только в том случае, если пространство выделенной связи
359     // ↪ предварительно заполнено нулями
360     if (link->Source != Constants.Null)
361     {
362         _sourcesTreeMethods.Detach(new IntPtr(&_header->FirstAsSource), linkIndex);
363     }
364     if (link->Target != Constants.Null)
365     {
366         _targetsTreeMethods.Detach(new IntPtr(&_header->FirstAsTarget), linkIndex);
367     }
368     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
369     var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
370     var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
371     if (leftTreeSize != rightTreeSize)
372     {
373         throw new Exception("One of the trees is broken.");
374     }
375     #endif
376     link->Source = values[Constants.SourcePart];
377     link->Target = values[Constants.TargetPart];
378     if (link->Source != Constants.Null)
379     {
380         _sourcesTreeMethods.Attach(new IntPtr(&_header->FirstAsSource), linkIndex);
381     }
382     if (link->Target != Constants.Null)
383     {
384         _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
385     }
386     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
387     leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
388     rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
389     if (leftTreeSize != rightTreeSize)
390     {
391         throw new Exception("One of the trees is broken.");
392     }
393     #endif
394     return linkIndex;
395 }
396
397 [MethodImpl(MethodImplOptions.AggressiveInlining)]
398 private IList<id> GetLinkStruct(id linkIndex)
399 {
400     var link = GetLinkUnsafe(linkIndex);
401     return new UInt64Link(linkIndex, link->Source, link->Target);
402 }
403
404 [MethodImpl(MethodImplOptions.AggressiveInlining)]
405 private Link* GetLinkUnsafe(id linkIndex) => &_amp;links[linkIndex];
406
407 /// <remarks>

```

```

406 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет пространство
407 /// </remarks>
408 public id Create()
409 {
410     var freeLink = _header->FirstFreeLink;
411     if (freeLink != Constants.Null)
412     {
413         _unusedLinksListMethods.Detach(freeLink);
414     }
415     else
416     {
417         if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
418         {
419             throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
420         }
421         if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
422         {
423             _memory.ReservedCapacity += _memory.ReservationStep;
424             SetPointers(_memory);
425             _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
426         }
427         _header->AllocatedLinks++;
428         _memory.UsedCapacity += sizeof(Link);
429         freeLink = _header->AllocatedLinks;
430     }
431     return freeLink;
432 }
433
434 public void Delete(id link)
435 {
436     if (link < _header->AllocatedLinks)
437     {
438         _unusedLinksListMethods.AttachAsFirst(link);
439     }
440     else if (link == _header->AllocatedLinks)
441     {
442         _header->AllocatedLinks--;
443         _memory.UsedCapacity -= sizeof(Link);
444         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор, пока не
445         //   ↳ дойдём до первой существующей связи
446         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
447         while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
448         {
449             _unusedLinksListMethods.Detach(_header->AllocatedLinks);
450             _header->AllocatedLinks--;
451             _memory.UsedCapacity -= sizeof(Link);
452         }
453     }
454 }
455
456 /// <remarks>
457 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если адрес
458 //   ↳ реально поменялся
459 ///
460 /// Указатель this.links может быть в том же месте,
461 /// так как 0-я связь не используется и имеет такой же размер как Header,
462 /// поэтому header размещается в том же месте, что и 0-я связь
463 /// </remarks>
464 private void SetPointers(IMemory memory)
465 {
466     if (memory == null)
467     {
468         _header = null;
469         _links = null;
470         _unusedLinksListMethods = null;
471         _targetsTreeMethods = null;
472         _unusedLinksListMethods = null;
473     }
474     else
475     {
476         _header = (LinksHeader*)(void*)memory.Pointer;
477         _links = (Link*)(void*)memory.Pointer;
478         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
479         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
480         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
481     }
482 }
483
484 [MethodImpl(MethodImplOptions.AggressiveInlining)]
485 private bool Exists(id link) => link >= Constants.MinPossibleIndex && link <=
486 //   ↳ _header->AllocatedLinks && !IsUnusedLink(link);
487
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
490 //   ↳ || (_links[link].SizeAsSource == Constants.Null &&
491 //   ↳   _links[link].Source != Constants.Null);

```

```

489     #region Disposable
490
491     protected override bool AllowMultipleDisposeCalls => true;
492
493     protected override void DisposeCore(bool manual, bool wasDisposed)
494     {
495         if (!wasDisposed)
496         {
497             SetPointers(null);
498         }
499         Disposable.TryDispose(_memory);
500     }
501
502     #endregion
503 }
504 }

```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs

```

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

```



```

31     protected abstract ulong GetBasePartValue(ulong link);
32
33     public ulong this[ulong index]
34     {
35         get
36         {
37             var root = GetTreeRoot();
38             if (index >= GetSize(root))
39             {
40                 return 0;
41             }
42             while (root != 0)
43             {
44                 var left = GetLeftOrDefault(root);
45                 var leftSize = GetSizeOrZero(left);
46                 if (index < leftSize)
47                 {
48                     root = left;
49                     continue;
50                 }
51                 if (index == leftSize)
52                 {
53                     return root;
54                 }
55                 root = GetRightOrDefault(root);
56                 index -= leftSize + 1;
57             }
58             return 0; // TODO: Impossible situation exception (only if tree structure broken)
59         }
60     }
61
62     // TODO: Return indices range instead of references count
63     public ulong CalculateReferences(ulong link)
64     {
65         var root = GetTreeRoot();
66         var total = GetSize(root);
67         var totalRightIgnore = OUL;
68         while (root != 0)
69         {
70             var @base = GetBasePartValue(root);
71             if (@base <= link)
72             {
73                 root = GetRightOrDefault(root);
74             }
75             else
76             {
77                 totalRightIgnore += GetRightSize(root) + 1;
78                 root = GetLeftOrDefault(root);
79             }
80         }
81         root = GetTreeRoot();
82         var totalLeftIgnore = OUL;
83         while (root != 0)
84         {
85             var @base = GetBasePartValue(root);
86             if (@base >= link)
87             {
88                 root = GetLeftOrDefault(root);
89             }
90             else
91             {
92                 totalLeftIgnore += GetLeftSize(root) + 1;
93                 root = GetRightOrDefault(root);
94             }
95         }
96         return total - totalRightIgnore - totalLeftIgnore;
97     }
98
99     public ulong EachReference(ulong link, Func<IList<ulong>, ulong> handler)
100     {
101         var root = GetTreeRoot();
102         if (root == 0)
103         {
104             return _constants.Continue;
105         }
106         ulong first = 0, current = root;
107         while (current != 0)
108         {
109             var @base = GetBasePartValue(current);
110             if (@base >= link)
111             {
112                 if (@base == link)
113                 {
114                     first = current;
115                 }
116                 current = GetLeftOrDefault(current);
117             }

```

```

118         else
119         {
120             current = GetRightOrDefault(current);
121         }
122     }
123     if (first != 0)
124     {
125         current = first;
126         while (true)
127         {
128             if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
129             {
130                 return _constants.Break;
131             }
132             current = GetNext(current);
133             if (current == 0 || GetBasePartValue(current) != link)
134             {
135                 break;
136             }
137         }
138     }
139     return _constants.Continue;
140 }
141
142 protected override void PrintNodeValue(ulong node, StringBuilder sb)
143 {
144     sb.Append(' ');
145     sb.Append(Links[node].Source);
146     sb.Append('-');
147     sb.Append('>');
148     sb.Append(Links[node].Target);
149 }
150
151 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
152 {
153     public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)
154         : base(memory)
155     {
156     }
157
158     protected override IntPtr GetLeftPointer(ulong node) => new
159         ↳ IntPtr(&Links[node].LeftAsSource);
160
161     protected override IntPtr GetRightPointer(ulong node) => new
162         ↳ IntPtr(&Links[node].RightAsSource);
163
164     protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsSource;
165     protected override ulong GetRightValue(ulong node) => Links[node].RightAsSource;
166
167     protected override ulong GetSize(ulong node)
168     {
169         var previousValue = Links[node].SizeAsSource;
170         //return MathHelpers.PartialRead(previousValue, 5, -5);
171         return (previousValue & 4294967264) >> 5;
172     }
173
174     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource = left;
175     protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
176         ↳ right;
177
178     protected override void SetSize(ulong node, ulong size)
179     {
180         var previousValue = Links[node].SizeAsSource;
181         //var modified = MathHelpers.PartialWrite(previousValue, size, 5, -5);
182         var modified = (previousValue & 31) | ((size & 134217727) << 5);
183         Links[node].SizeAsSource = modified;
184     }
185
186     protected override bool GetLeftIsChild(ulong node)
187     {
188         var previousValue = Links[node].SizeAsSource;
189         //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
190         return (previousValue & 16) >> 4 == 1UL;
191     }
192
193     protected override void SetLeftIsChild(ulong node, bool value)
194     {
195         var previousValue = Links[node].SizeAsSource;
196         //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
197         var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
198         Links[node].SizeAsSource = modified;
199     }
200
201     protected override bool GetRightIsChild(ulong node)

```

```

202 {
203     var previousValue = Links[node].SizeAsSource;
204     //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
205     return (previousValue & 8) >> 3 == 1UL;
206 }
207
208 protected override void SetRightIsChild(ulong node, bool value)
209 {
210     var previousValue = Links[node].SizeAsSource;
211     //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
212     var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
213     Links[node].SizeAsSource = modified;
214 }
215
216 protected override sbyte GetBalance(ulong node)
217 {
218     var previousValue = Links[node].SizeAsSource;
219     //var value = MathHelpers.PartialRead(previousValue, 0, 3);
220     var value = previousValue & 7;
221     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 | 124 :
222         ↪ value & 3);
223     return unpackedValue;
224 }
225
226 protected override void SetBalance(ulong node, sbyte value)
227 {
228     var previousValue = Links[node].SizeAsSource;
229     var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
230     //var modified = MathHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
231     var modified = (previousValue & 4294967288) | (packagedValue & 7);
232     Links[node].SizeAsSource = modified;
233 }
234
235 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
236     => Links[first].Source < Links[second].Source ||
237     (Links[first].Source == Links[second].Source && Links[first].Target <
238     ↪ Links[second].Target);
239
240 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
241     => Links[first].Source > Links[second].Source ||
242     (Links[first].Source == Links[second].Source && Links[first].Target >
243     ↪ Links[second].Target);
244
245 protected override ulong GetTreeRoot() => Header->FirstAsSource;
246
247 protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
248
249 /// <summary>
250 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
251 ↪ (концом)
252 /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
253 /// </summary>
254 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
255 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
256 /// <returns>Индекс искомой связи.</returns>
257 public ulong Search(ulong source, ulong target)
258 {
259     var root = Header->FirstAsSource;
260     while (root != 0)
261     {
262         var rootSource = Links[root].Source;
263         var rootTarget = Links[root].Target;
264         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) // node.Key <
265             ↪ root.Key
266         {
267             root = GetLeftOrDefault(root);
268         }
269         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
270             ↪ node.Key > root.Key
271         {
272             root = GetRightOrDefault(root);
273         }
274         else // node.Key == root.Key
275         {
276             return root;
277         }
278     }
279     return 0;
280 }
281
282 [MethodImpl(MethodImplOptions.AggressiveInlining)]
283 private static bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget, ulong
284     ↪ secondSource, ulong secondTarget)
285     => firstSource < secondSource || (firstSource == secondSource && firstTarget <
286     ↪ secondTarget);
287
288 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

281 private static bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget, ulong
    ↳ secondSource, ulong secondTarget)
282     => firstSource > secondSource || (firstSource == secondSource && firstTarget >
    ↳ secondTarget);
283
284 [MethodImpl(MethodImplOptions.AggressiveInlining)]
285 protected override void ClearNode(ulong node)
286 {
287     Links[node].LeftAsSource = OUL;
288     Links[node].RightAsSource = OUL;
289     Links[node].SizeAsSource = OUL;
290 }
291
292 [MethodImpl(MethodImplOptions.AggressiveInlining)]
293 protected override ulong GetZero() => OUL;
294
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) => *(ulong*)pointer.ToPointer() ==
    ↳ OUL;
303
304 [MethodImpl(MethodImplOptions.AggressiveInlining)]
305 protected override bool EqualToZero(ulong value) => value == OUL;
306
307 [MethodImpl(MethodImplOptions.AggressiveInlining)]
308 protected override bool IsEquals(ulong first, ulong second) => first == second;
309
310 [MethodImpl(MethodImplOptions.AggressiveInlining)]
311 protected override bool GreaterThanZero(ulong value) => value > OUL;
312
313 [MethodImpl(MethodImplOptions.AggressiveInlining)]
314 protected override bool GreaterThan(ulong first, ulong second) => first > second;
315
316 [MethodImpl(MethodImplOptions.AggressiveInlining)]
317 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
318
319 [MethodImpl(MethodImplOptions.AggressiveInlining)]
320 protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
    ↳ always true for ulong
321
322 [MethodImpl(MethodImplOptions.AggressiveInlining)]
323 protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is always
    ↳ >= 0 for ulong
324
325 [MethodImpl(MethodImplOptions.AggressiveInlining)]
326 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
327
328 [MethodImpl(MethodImplOptions.AggressiveInlining)]
329 protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
    ↳ for ulong
330
331 [MethodImpl(MethodImplOptions.AggressiveInlining)]
332 protected override bool LessThan(ulong first, ulong second) => first < second;
333
334 [MethodImpl(MethodImplOptions.AggressiveInlining)]
335 protected override ulong Increment(ulong value) => ++value;
336
337 [MethodImpl(MethodImplOptions.AggressiveInlining)]
338 protected override ulong Decrement(ulong value) => --value;
339
340 [MethodImpl(MethodImplOptions.AggressiveInlining)]
341 protected override ulong Add(ulong first, ulong second) => first + second;
342
343 [MethodImpl(MethodImplOptions.AggressiveInlining)]
344 protected override ulong Subtract(ulong first, ulong second) => first - second;
345 }
346
347 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
348 {
349     public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
350         : base(memory)
351     {
352     }
353
354     //protected override IntPtr GetLeft(ulong node) => new IntPtr(&Links[node].LeftAsTarget);
355     //protected override IntPtr GetRight(ulong node) => new IntPtr(&Links[node].RightAsTarget);
356     //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
357     //protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
    ↳ left;
358
359
360
361

```

```

362 //protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
    ↳ right;
363
364 //protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
    ↳ size;
365
366 protected override IntPtr GetLeftPointer(ulong node) => new
    ↳ IntPtr(&Links[node].LeftAsTarget);
367
368 protected override IntPtr GetRightPointer(ulong node) => new
    ↳ IntPtr(&Links[node].RightAsTarget);
369
370 protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsTarget;
371
372 protected override ulong GetRightValue(ulong node) => Links[node].RightAsTarget;
373
374 protected override ulong GetSize(ulong node)
375 {
376     var previousValue = Links[node].SizeAsTarget;
377     //return MathHelpers.PartialRead(previousValue, 5, -5);
378     return (previousValue & 4294967264) >> 5;
379 }
380
381 protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget = left;
382
383 protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
    ↳ right;
384
385 protected override void SetSize(ulong node, ulong size)
386 {
387     var previousValue = Links[node].SizeAsTarget;
388     //var modified = MathHelpers.PartialWrite(previousValue, size, 5, -5);
389     var modified = (previousValue & 31) | ((size & 134217727) << 5);
390     Links[node].SizeAsTarget = modified;
391 }
392
393 protected override bool GetLeftIsChild(ulong node)
394 {
395     var previousValue = Links[node].SizeAsTarget;
396     //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
397     return (previousValue & 16) >> 4 == 1UL;
398     // TODO: Check if this is possible to use
399     //var nodeSize = GetSize(node);
400     //var left = GetLeftValue(node);
401     //var leftSize = GetSizeOrZero(left);
402     //return leftSize > 0 && nodeSize > leftSize;
403 }
404
405 protected override void SetLeftIsChild(ulong node, bool value)
406 {
407     var previousValue = Links[node].SizeAsTarget;
408     //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
409     var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
410     Links[node].SizeAsTarget = modified;
411 }
412
413 protected override bool GetRightIsChild(ulong node)
414 {
415     var previousValue = Links[node].SizeAsTarget;
416     //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
417     return (previousValue & 8) >> 3 == 1UL;
418     // TODO: Check if this is possible to use
419     //var nodeSize = GetSize(node);
420     //var right = GetRightValue(node);
421     //var rightSize = GetSizeOrZero(right);
422     //return rightSize > 0 && nodeSize > rightSize;
423 }
424
425 protected override void SetRightIsChild(ulong node, bool value)
426 {
427     var previousValue = Links[node].SizeAsTarget;
428     //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
429     var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
430     Links[node].SizeAsTarget = modified;
431 }
432
433 protected override sbyte GetBalance(ulong node)
434 {
435     var previousValue = Links[node].SizeAsTarget;
436     //var value = MathHelpers.PartialRead(previousValue, 0, 3);
437     var value = previousValue & 7;
438     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 | 124 :
    ↳ value & 3);
439     return unpackedValue;
440 }
441
442 protected override void SetBalance(ulong node, sbyte value)

```

```

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

```

# ./Sequences/Converters/BalancedVariantConverter.cs

```

1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Sequences.Converters
4 {
5     public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
6     {
7         public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
8
9         public override TLink Convert(IList<TLink> sequence)
10         {
11             var length = sequence.Count;
12             if (length < 1)
13             {
14                 return default;
15             }
16             if (length == 1)
17             {
18                 return sequence[0];
19             }
20             // Make copy of next layer
21             if (length > 2)
22             {
23                 // TODO: Try to use stackalloc (which at the moment is not working with generics) but
24                 ↪ 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         private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
40         {
41             var loopedLength = length - (length % 2);
42             for (var i = 0; i < loopedLength; i += 2)
43             {
44                 destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
45             }
46             if (length > loopedLength)
47             {
48                 destination[length / 2] = source[length - 1];
49             }
50         }
51 }

```

## ./Sequences/Converters/CompressingConverter.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Collections;
6  using Platform.Helpers.Singletons;
7  using Platform.Numbers;
8  using Platform.Data.Constants;
9  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
10
11 namespace Platform.Data.Doublets.Sequences.Converters
12 {
13     /// <remarks>
14     /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от Links на
15     /// этапе сжатия.
16     /// А именно будет создаваться временный список пар необходимых для выполнения сжатия, в таком
17     /// 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 пар, а
19     /// 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> baseConverter,
51             ↳ 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> baseConverter,
57             ↳ LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool doInitialFrequenciesIncrement)
58             : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
59                 ↳ doInitialFrequenciesIncrement)
60         {
61         }
62
63         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink> baseConverter,
64             ↳ LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink minFrequencyToCompress, bool
65             ↳ doInitialFrequenciesIncrement)
66             : base(links)
67         {
68             _baseConverter = baseConverter;
69             _doubletFrequenciesCache = doubletFrequenciesCache;
70             if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
71             {
72                 minFrequencyToCompress = Integer<TLink>.One;
73             }
74             _minFrequencyToCompress = minFrequencyToCompress;
75             _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
76             ResetMaxDoublet();
77         }
78
79         public override TLink Convert(IList<TLink> source) => _baseConverter.Convert(Compress(source));
80
81         /// <remarks>
82         /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
83         /// Faster version (doublets' frequencies dictionary is not recreated).
84         /// </remarks>
85         private IList<TLink> Compress(IList<TLink> sequence)
86         {
87             if (sequence.IsNullOrEmpty())
88             {
89             }
90         }
91     }
92 }
```

```

79         return null;
80     }
81     if (sequence.Count == 1)
82     {
83         return sequence;
84     }
85     if (sequence.Count == 2)
86     {
87         return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
88     }
89     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
90     var copy = new HalfDoublet[sequence.Count];
91     Doublet<TLink> doublet = default;
92     for (var i = 1; i < sequence.Count; i++)
93     {
94         doublet.Source = sequence[i - 1];
95         doublet.Target = sequence[i];
96         LinkFrequency<TLink> data;
97         if (_doInitialFrequenciesIncrement)
98         {
99             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
100         }
101         else
102         {
103             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
104             if (data == null)
105             {
106                 throw new NotSupportedException("If you ask not to increment frequencies, it is
107                     ↪ expected that all frequencies for the sequence are prepared.");
108             }
109             copy[i - 1].Element = sequence[i - 1];
110             copy[i - 1].DoubletData = data;
111             UpdateMaxDoublet(ref doublet, data);
112         }
113         copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
114         copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
115         if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
116         {
117             var newLength = ReplaceDoublets(copy);
118             sequence = new TLink[newLength];
119             for (int i = 0; i < newLength; i++)
120             {
121                 sequence[i] = copy[i].Element;
122             }
123         }
124         return sequence;
125     }
126
127     /// <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();

```



```

161         copy[w].DoubletData =
            ↳ _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
            ↳ next);
162     }
163     copy[w++].Element = maxDoubletReplacementLink;
164     r++;
165     newLength--;
166 }
167 else
168 {
169     copy[w++] = copy[r];
170 }
171 }
172 if (w < newLength)
173 {
174     copy[w] = copy[r];
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 || (maxFrequency ==
208     ↳ frequency && doublet.Source + doublet.Target < /* gives better compression string data
209     ↳ (and gives collisions quickly) */ _maxDoublet.Source + _maxDoublet.Target)))
210     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
211         (_comparer.Compare(maxFrequency, frequency) < 0 ||
212         ↳ (_equalityComparer.Equals(maxFrequency, frequency) &&
213         ↳ _comparer.Compare(ArithmeticHelpers.Add(doublet.Source, doublet.Target),
214         ↳ ArithmeticHelpers.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
215         ↳ better stability and better compression on sequent data and even on random numbers
216         ↳ data (but gives collisions anyway) */
217     {
218         _maxDoublet = doublet;
219         _maxDoubletData = data;
220     }
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>, TLink>
7      {
8          protected readonly ILinks<TLink> Links;
9          public LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
10         public abstract TLink Convert(IList<TLink> source);
11     }
12 }

```

#### ./Sequences/Converters/OptimalVariantConverter.cs

```

1  using System.Collections.Generic;
2  using System.Linq;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Sequences.Converters
6  {

```

```

7 public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
8 {
9     private static readonly EqualityComparer<TLink> _equalityComparer =
10         EqualityComparer<TLink>.Default;
11     private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13     private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
14
15     public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
16         sequenceToItsLocalElementLevelsConverter) : base(links)
17     => _sequenceToItsLocalElementLevelsConverter = sequenceToItsLocalElementLevelsConverter;
18
19     public override TLink Convert(IList<TLink> sequence)
20     {
21         var length = sequence.Count;
22         if (length == 1)
23         {
24             return sequence[0];
25         }
26         var links = Links;
27         if (length == 2)
28         {
29             return links.GetOrCreate(sequence[0], sequence[1]);
30         }
31         sequence = sequence.ToArray();
32         var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
33         while (length > 2)
34         {
35             var levelRepeat = 1;
36             var currentLevel = levels[0];
37             var previousLevel = levels[0];
38             var skipOnce = false;
39             var w = 0;
40             for (var i = 1; i < length; i++)
41             {
42                 if (_equalityComparer.Equals(currentLevel, levels[i]))
43                 {
44                     levelRepeat++;
45                     skipOnce = false;
46                     if (levelRepeat == 2)
47                     {
48                         sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
49                         var newLevel = i >= length - 1 ?
50                             GetPreviousLowerThanCurrentOrCurrent(previousLevel, currentLevel) :
51                             i < 2 ?
52                             GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
53                             GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
54                                 currentLevel, levels[i + 1]);
55                         levels[w] = newLevel;
56                         previousLevel = currentLevel;
57                         w++;
58                         levelRepeat = 0;
59                         skipOnce = true;
60                     }
61                     else if (i == length - 1)
62                     {
63                         sequence[w] = sequence[i];
64                         levels[w] = levels[i];
65                         w++;
66                     }
67                 }
68                 else
69                 {
70                     currentLevel = levels[i];
71                     levelRepeat = 1;
72                     if (skipOnce)
73                     {
74                         skipOnce = false;
75                     }
76                     else
77                     {
78                         sequence[w] = sequence[i - 1];
79                         levels[w] = levels[i - 1];
80                         previousLevel = levels[w];
81                         w++;
82                     }
83                     if (i == length - 1)
84                     {
85                         sequence[w] = sequence[i];
86                         levels[w] = levels[i];
87                         w++;
88                     }
89                 }
90             }
91             length = w;
92         }
93         return links.GetOrCreate(sequence[0], sequence[1]);
94     }
95 }

```

```

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

```

#### ./Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs

```

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

```

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

```

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

```

#### ./Sequences/CreteriaMatchers/MarkedSequenceCreteriaMatcher.cs

```

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

```

```

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

```

```

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

```

```

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

```

```

16 public class DuplicateSegmentsProvider<TLink> : DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
    ↳ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
17 {
18     private readonly ILinks<TLink> _links;
19     private readonly ISequences<TLink> _sequences;
20     private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
21     private BitString _visited;
22
23     private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
24     {
25         private readonly IListEqualityComparer<TLink> _listComparer;
26         public ItemEquilityComparer() => _listComparer =
            ↳ Default<IListEqualityComparer<TLink>>.Instance;
27         public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
            ↳ KeyValuePair<IList<TLink>, IList<TLink>> right) => _listComparer.Equals(left.Key,
            ↳ right.Key) && _listComparer.Equals(left.Value, right.Value);
28         public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
            ↳ HashHelpers.Generate(_listComparer.GetHashCode(pair.Key),
            ↳ _listComparer.GetHashCode(pair.Value));
29     }
30
31     private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
32     {
33         private readonly IListComparer<TLink> _listComparer;
34
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(ILinks<TLink> links, ISequences<TLink> sequences)
49         : base(minimumStringSegmentLength: 2)
50     {
51         _links = links;
52         _sequences = sequences;
53     }
54
55     public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
56     {
57         _groups = new HashSet<KeyValuePair<IList<TLink>,
            ↳ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
58         var count = _links.Count();
59         _visited = new BitString((long)(Integer<TLink>)count + 1);
60         _links.Each(link =>
61         {
62             var linkIndex = _links.GetIndex(link);
63             var linkBitIndex = (long)(Integer<TLink>)linkIndex;
64             if (!_visited.Get(linkBitIndex))
65             {
66                 var sequenceElements = new List<TLink>();
67                 _sequences.EachPart(sequenceElements.AddAndReturnTrue, linkIndex);
68                 if (sequenceElements.Count > 2)
69                 {
70                     WalkAll(sequenceElements);
71                 }
72             }
73             return _links.Constants.Continue;
74         });
75         var resultList = _groups.ToList();
76         var comparer = Default<ItemComparer>.Instance;
77         resultList.Sort(comparer);
78
79         #if DEBUG
80         foreach (var item in resultList)
81         {
82             PrintDuplicates(item);
83         }
84         #endif
85         return resultList;
86     }
87
88     protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int length)
89         ↳ => new Segment<TLink>(elements, offset, length);
90
91     protected override void OnDuplicateFound(Segment<TLink> segment)
92     {
93         var duplicates = CollectDuplicatesForSegment(segment);
94         if (duplicates.Count > 1)
95         {

```

```

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

```

./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs

```

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

```

./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToItsFrequencyNumberConverter.cs

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

./Sequences/Frequencies/Cache/LinkFrequenciesCache.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Interfaces;
5 using Platform.Numbers;
6
7 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
8 {
9     /// <remarks>
10     /// Can be used to operate with many CompressingConverters (to keep global frequencies data between
11     /// them).
12     /// TODO: Extract interface to implement frequencies storage inside Links storage
13     /// </remarks>
14     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
15     {
16         private static readonly EqualityComparer<TLink> _equalityComparer =
17             ↳ EqualityComparer<TLink>.Default;
18         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19
20         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
21         private readonly ICounter<TLink, TLink> _frequencyCounter;
22
23         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
24             : base(links)
25         {
26             _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
27                 ↳ DoubletComparer<TLink>.Default);
28             _frequencyCounter = frequencyCounter;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
33         {
34             var doublet = new Doublet<TLink>(source, target);
35             return GetFrequency(ref doublet);
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
40         {
41             _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
42             return data;
43         }
44
45         public void IncrementFrequencies(IList<TLink> sequence)
46         {
47             for (var i = 1; i < sequence.Count; i++)
48             {
49                 IncrementFrequency(sequence[i - 1], sequence[i]);
50             }
51         }
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
55         {
56             var doublet = new Doublet<TLink>(source, target);
57             return IncrementFrequency(ref doublet);
58         }
59
60         public void PrintFrequencies(IList<TLink> sequence)
61         {
62             for (var i = 1; i < sequence.Count; i++)
63             {
64                 PrintFrequency(sequence[i - 1], sequence[i]);
65             }
66         }
67
68         public void PrintFrequency(TLink source, TLink target)
69         {
70             var number = GetFrequency(source, target).Frequency;
71             Console.WriteLine("{0},{1} - {2}", source, target, number);
72         }
73     }
74 }
```

```

70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
72 {
73     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
74     {
75         data.IncrementFrequency();
76     }
77     else
78     {
79         var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
80         data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
81         if (!_equalityComparer.Equals(link, default))
82         {
83             data.Frequency = ArithmeticHelpers.Add(data.Frequency,
84                 ↪ _frequencyCounter.Count(link));
85         }
86         _doubletsCache.Add(doublet, data);
87     }
88     return data;
89 }
90
91 public void ValidateFrequencies()
92 {
93     foreach (var entry in _doubletsCache)
94     {
95         var value = entry.Value;
96         var linkIndex = value.Link;
97         if (!_equalityComparer.Equals(linkIndex, default))
98         {
99             var frequency = value.Frequency;
100             var count = _frequencyCounter.Count(linkIndex);
101             // TODO: Why `frequency` always greater than `count` by 1?
102             if (((_comparer.Compare(frequency, count) > 0) &&
103                 ↪ (_comparer.Compare(ArithmeticHelpers.Subtract(frequency, count),
104                 ↪ Integer<TLink>.One) > 0))
105                 || ((_comparer.Compare(count, frequency) > 0) &&
106                 ↪ (_comparer.Compare(ArithmeticHelpers.Subtract(count, frequency),
107                 ↪ Integer<TLink>.One) > 0)))
108             {
109                 throw new InvalidOperationException("Frequencies validation failed.");
110             }
111             //else
112             //{
113             //    if (value.Frequency > 0)
114             //    {
115             //        var frequency = value.Frequency;
116             //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
117             //        var count = _countLinkFrequency(linkIndex);
118             //        if ((frequency > count && frequency - count > 1) || (count > frequency &&
119             //            ↪ count - frequency > 1))
120             //            throw new Exception("Frequencies validation failed.");
121             //    }
122             //}
123     }
124 }

```

#### ./Sequences/Frequencies/Cache/LinkFrequency.cs

```

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

```



```

26     }
27 }

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

```

```

./Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4  using Platform.Data.Sequences;
5
6  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7  {
8      public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
        ↳ EqualityComparer<TLink>.Default;
11         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13         protected readonly ILinks<TLink> _links;
14         protected readonly TLink _sequenceLink;
15         protected readonly TLink _symbol;
16         protected TLink _total;
17
18         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink, TLink
        ↳ symbol)
19         {
20             _links = links;
21             _sequenceLink = sequenceLink;
22             _symbol = symbol;
23             _total = default;
24         }
25
26         public virtual TLink Count()
27         {
28             if (_comparer.Compare(_total, default) > 0)
29             {
30                 return _total;
31             }
32             StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
        ↳ IsElement, VisitElement);
33             return _total;
34         }
35
36         private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
        ↳ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
        ↳ IsPartialPoint
37
38         private bool VisitElement(TLink element)
39         {
40             if (_equalityComparer.Equals(element, _symbol))
41             {
42                 _total = ArithmeticHelpers.Increment(_total);
43             }
44             return true;
45         }
46     }
47 }

```

```

./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs
1  using Platform.Interfaces;
2
3  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
4  {

```

```

5     public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
6     {
7         private readonly ILinks<TLink> _links;
8         private readonly ICriteriaMatcher<TLink> _markedSequenceMatcher;
9
10        public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links, ICriteriaMatcher<TLink>
11        ↪ markedSequenceMatcher)
12        {
13            _links = links;
14            _markedSequenceMatcher = markedSequenceMatcher;
15        }
16
17        public TLink Count(TLink argument) => new
18        ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links, _markedSequenceMatcher,
19        ↪ argument).Count();
20    }
21 }

```

#### ./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs

```

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

```

#### ./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs

```

1 using Platform.Interfaces;
2
3 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
4 {
5     public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, 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        }
33    }
34 }

```

```

31     CountCore(_symbol);
32     return _total;
33 }
34
35 private void CountCore(TLink link)
36 {
37     var any = _links.Constants.Any;
38     if (_equalityComparer.Equals(_links.Count(any, link), default))
39     {
40         CountSequenceSymbolFrequency(link);
41     }
42     else
43     {
44         _links.Each(EachElementHandler, any, link);
45     }
46 }
47
48 protected virtual void CountSequenceSymbolFrequency(TLink link)
49 {
50     var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links, link,
51     ↪ _symbol);
52     _total = ArithmeticHelpers.Add(_total, symbolFrequencyCounter.Count());
53 }
54
55 private TLink EachElementHandler(ICollection<TLink> doublet)
56 {
57     var constants = _links.Constants;
58     var doubletIndex = doublet[constants.IndexPart];
59     if (_visits.Add(doubletIndex))
60     {
61         CountCore(doubletIndex);
62     }
63     return constants.Continue;
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             ICollection<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>
8     {
9         private readonly ICriteriaMatcher<TLink> _elementMatcher;
10
11         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriteriaMatcher<TLink>
12             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
13
14         public TLink Get(TLink sequence)
15         {
16             var height = default(TLink);
17             var pairOrElement = sequence;
18             while (!_elementMatcher.IsMatched(pairOrElement))
19             {
20                 pairOrElement = Links.GetTarget(pairOrElement);
21                 height = ArithmeticHelpers.Increment(height);
22             }
23             return height;
24         }
25     }
26 }
```

## ./Sequences/HeightProviders/ISequenceHeightProvider.cs

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

## ./Sequences/Sequences.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Runtime.CompilerServices;
5 using Platform.Collections;
6 using Platform.Collections.Lists;
7 using Platform.Threading.Synchronization;
8 using Platform.Helpers.Singletons;
9 using LinkIndex = System.UInt64;
10 using Platform.Data.Constants;
11 using Platform.Data.Sequences;
12 using Platform.Data.Doublets.Sequences.Walkers;
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     /// <summary>
17     /// Представляет коллекцию последовательностей связей.
18     /// </summary>
19     /// <remarks>
20     /// Обязательно реализовать атомарность каждого публичного метода.
21     ///
22     /// TODO:
23     ///
24     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
25     /// через естественную группировку по unicode типам, все whitespace вместе, все символы вместе, все
26     /// ↳ числа вместе и т.п.
27     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину графа)
28     ///
29     /// х*у - найти все связи между, в последовательностях любой формы, если не стоит ограничитель на
30     /// ↳ то, что является последовательностью, а что нет,
31     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком порядке.
32     ///
33     /// Рост последовательности слева и справа.
34     /// Поиск со звёздочкой.
35     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
36     /// так же проблема может быть решена при реализации дистанционных триггеров.
37     /// Нужны ли уникальные указатели вообще?
38     /// Что если обращение к информации будет происходить через содержимое всегда?
39     ///
40     /// Писать тесты.
41     ///
42     ///
43     /// Можно убрать зависимость от конкретной реализации Links,
44     /// на зависимость от абстрактного элемента, который может быть представлен несколькими способами.
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     }
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 Id.
261     Func<ulong, bool> innerHandler = Options.UseSequenceMarker ? (Func<ulong,
262     ↪ bool>)matcher.HandleFullMatchedSequence : matcher.HandleFullMatched;
263     //if (sequence.Length >= 2)
264     if (!StepRight(innerHandler, sequence[0], sequence[1]))
265     {
266         return false;
267     }
268     var last = sequence.Count - 2;
269     for (var i = 1; i < last; i++)
270     {
271         if (!PartialStepRight(innerHandler, sequence[i], sequence[i + 1]))
272         {
273             return false;
274         }
275     }
276     if (sequence.Count >= 3)
277     {
278         if (!StepLeft(innerHandler, sequence[sequence.Count - 2], sequence[sequence.Count - 1]))
279         {
280             return false;
281         }
282     }
283     return true;
284 }
285
286 private bool PartialStepRight(Func<ulong, bool> handler, ulong left, ulong right)
287 {
288     return Links.Unsync.Each(_constants.Any, left, doublet =>
289     {
290         if (!StepRight(handler, doublet, right))
291         {
292             return false;
293         }
294         if (left != doublet)
295         {
296             return PartialStepRight(handler, doublet, right);
297         }
298         return true;
299     });
300 }
301
302 private bool StepRight(Func<ulong, bool> handler, ulong left, ulong right) =>
303     ↪ Links.Unsync.Each(left, _constants.Any, rightStep => TryStepRightUp(handler, right,
304     ↪ rightStep));
305
306 private bool TryStepRightUp(Func<ulong, bool> handler, ulong right, ulong stepFrom)
307 {

```

```

305     var upStep = stepFrom;
306     var firstSource = Links.Unsync.GetTarget(upStep);
307     while (firstSource != right && firstSource != upStep)
308     {
309         upStep = firstSource;
310         firstSource = Links.Unsync.GetSource(upStep);
311     }
312     if (firstSource == right)
313     {
314         return handler(stepFrom);
315     }
316     return true;
317 }
318
319 private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong right) =>
    ↳ Links.Unsync.Each(_constants.Any, right, leftStep => TryStepLeftUp(handler, left,
    ↳ leftStep));
320
321 private bool TryStepLeftUp(Func<ulong, bool> handler, ulong left, ulong stepFrom)
322 {
323     var upStep = stepFrom;
324     var firstTarget = Links.Unsync.GetSource(upStep);
325     while (firstTarget != left && firstTarget != upStep)
326     {
327         upStep = firstTarget;
328         firstTarget = Links.Unsync.GetTarget(upStep);
329     }
330     if (firstTarget == left)
331     {
332         return handler(stepFrom);
333     }
334     return true;
335 }
336
337 #endregion
338
339 #region Update
340
341 public ulong Update(ulong[] sequence, ulong[] newSequence)
342 {
343     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
344     {
345         return _constants.Null;
346     }
347     if (sequence.IsNullOrEmpty())
348     {
349         return Create(newSequence);
350     }
351     if (newSequence.IsNullOrEmpty())
352     {
353         Delete(sequence);
354         return _constants.Null;
355     }
356     return Sync.ExecuteWriteOperation(() =>
357     {
358         Links.EnsureEachLinkIsAnyOrExists(sequence);
359         Links.EnsureEachLinkExists(newSequence);
360         return UpdateCore(sequence, newSequence);
361     });
362 }
363
364 private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
365 {
366     ulong bestVariant;
367     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew && !sequence.EqualTo(newSequence))
368     {
369         bestVariant = CompactCore(newSequence);
370     }
371     else
372     {
373         bestVariant = CreateCore(newSequence);
374     }
375     // TODO: Check all options only ones before loop execution
376     // Возможно нужно две версии Each, возвращающий фактические последовательности и с маркером,
377     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты можно
378     ↳ получить имея только фактические последовательности.
379     foreach (var variant in Each(sequence))
380     {
381         if (variant != bestVariant)
382         {
383             UpdateOneCore(variant, bestVariant);
384         }
385     }
386     return bestVariant;
387 }
388
389 private void UpdateOneCore(ulong sequence, ulong newSequence)

```



```

389 {
390     if (Options.UseGarbageCollection)
391     {
392         var sequenceElements = GetSequenceElements(sequence);
393         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
394         var sequenceLink = GetSequenceByElements(sequenceElements);
395         var newSequenceElements = GetSequenceElements(newSequence);
396         var newSequenceLink = GetSequenceByElements(newSequenceElements);
397         if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
398         {
399             if (sequenceLink != _constants.Null)
400             {
401                 Links.Unsync.Merge(sequenceLink, newSequenceLink);
402             }
403             Links.Unsync.Merge(sequenceElements, newSequenceElements);
404         }
405         ClearGarbage(sequenceElementsContents.Source);
406         ClearGarbage(sequenceElementsContents.Target);
407     }
408     else
409     {
410         if (Options.UseSequenceMarker)
411         {
412             var sequenceElements = GetSequenceElements(sequence);
413             var sequenceLink = GetSequenceByElements(sequenceElements);
414             var newSequenceElements = GetSequenceElements(newSequence);
415             var newSequenceLink = GetSequenceByElements(newSequenceElements);
416             if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
417             {
418                 if (sequenceLink != _constants.Null)
419                 {
420                     Links.Unsync.Merge(sequenceLink, newSequenceLink);
421                 }
422                 Links.Unsync.Merge(sequenceElements, newSequenceElements);
423             }
424         }
425         else
426         {
427             if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
428             {
429                 Links.Unsync.Merge(sequence, newSequence);
430             }
431         }
432     }
433 }
434
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 определить
527 /// ↳ 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 }

```

```

562 public class Matcher : RightSequenceWalker<ulong>
563 {
564     private readonly Sequences _sequences;
565     private readonly IList<LinkIndex> _patternSequence;
566     private readonly HashSet<LinkIndex> _linksInSequence;
567     private readonly HashSet<LinkIndex> _results;
568     private readonly Func<ulong, bool> _stopableHandler;
569     private readonly HashSet<ulong> _readAsElements;
570     private int _filterPosition;
571
572     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence, HashSet<LinkIndex>
573         ↪ results, Func<LinkIndex, bool> stopableHandler, HashSet<LinkIndex> readAsElements =
574         ↪ null)
575         : base(sequences.Links.Unsync)
576     {
577         _sequences = sequences;
578         _patternSequence = patternSequence;
579         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
580             ↪ _constants.Any && x != ZeroOrMany));
581         _results = results;
582         _stopableHandler = stopableHandler;
583         _readAsElements = readAsElements;
584     }
585
586     protected override bool IsElement(IList<ulong> link) => base.IsElement(link) ||
587         ↪ (_readAsElements != null && _readAsElements.Contains(Links.GetIndex(link))) ||
588         ↪ _linksInSequence.Contains(Links.GetIndex(link));
589
590     public bool FullMatch(LinkIndex sequenceToMatch)
591     {
592         _filterPosition = 0;
593         foreach (var part in Walk(sequenceToMatch))
594         {
595             if (!FullMatchCore(Links.GetIndex(part)))
596             {
597                 break;
598             }
599         }
600         return _filterPosition == _patternSequence.Count;
601     }
602
603     private bool FullMatchCore(LinkIndex element)
604     {
605         if (_filterPosition == _patternSequence.Count)
606         {
607             _filterPosition = -2; // Длиннее чем нужно
608             return false;
609         }
610         if (_patternSequence[_filterPosition] != _constants.Any
611             && element != _patternSequence[_filterPosition])
612         {
613             _filterPosition = -1;
614             return false; // Начинается/Продолжается иначе
615         }
616         _filterPosition++;
617         return true;
618     }
619
620     public void AddFullMatchedToResults(ulong sequenceToMatch)
621     {
622         if (FullMatch(sequenceToMatch))
623         {
624             _results.Add(sequenceToMatch);
625         }
626     }
627
628     public bool HandleFullMatched(ulong sequenceToMatch)
629     {
630         if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
631         {
632             return _stopableHandler(sequenceToMatch);
633         }
634         return true;
635     }
636
637     public bool HandleFullMatchedSequence(ulong sequenceToMatch)
638     {
639         var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
640         if (sequence != _constants.Null && FullMatch(sequenceToMatch) &&
641             ↪ _results.Add(sequenceToMatch))
642         {
643             return _stopableHandler(sequence);
644         }
645         return true;
646     }
647
648     /// <remarks>

```

```

644 /// TODO: Add support for LinksConstants.Any
645 /// </remarks>
646 public bool PartialMatch(LinkIndex sequenceToMatch)
647 {
648     _filterPosition = -1;
649     foreach (var part in Walk(sequenceToMatch))
650     {
651         if (!PartialMatchCore(Links.GetIndex(part)))
652         {
653             break;
654         }
655     }
656     return _filterPosition == _patternSequence.Count - 1;
657 }
658
659 private bool PartialMatchCore(LinkIndex element)
660 {
661     if (_filterPosition == (_patternSequence.Count - 1))
662     {
663         return false; // Нашлось
664     }
665     if (_filterPosition >= 0)
666     {
667         if (element == _patternSequence[_filterPosition + 1])
668         {
669             _filterPosition++;
670         }
671         else
672         {
673             _filterPosition = -1;
674         }
675     }
676     if (_filterPosition < 0)
677     {
678         if (element == _patternSequence[0])
679         {
680             _filterPosition = 0;
681         }
682     }
683     return true; // Ищем дальше
684 }
685
686 public void AddPartialMatchedToResults(ulong sequenceToMatch)
687 {
688     if (PartialMatch(sequenceToMatch))
689     {
690         _results.Add(sequenceToMatch);
691     }
692 }
693
694 public bool HandlePartialMatched(ulong sequenceToMatch)
695 {
696     if (PartialMatch(sequenceToMatch))
697     {
698         return _stopableHandler(sequenceToMatch);
699     }
700     return true;
701 }
702
703 public void AddAllPartialMatchedToResults(IEnumerable<ulong> sequencesToMatch)
704 {
705     foreach (var sequenceToMatch in sequencesToMatch)
706     {
707         if (PartialMatch(sequenceToMatch))
708         {
709             _results.Add(sequenceToMatch);
710         }
711     }
712 }
713
714 public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<ulong>
715 ↪ sequencesToMatch)
716 {
717     foreach (var sequenceToMatch in sequencesToMatch)
718     {
719         if (PartialMatch(sequenceToMatch))
720         {
721             _readAsElements.Add(sequenceToMatch);
722             _results.Add(sequenceToMatch);
723         }
724     }
725 }
726
727 #endregion
728 }
729 }

```

## ./Sequences/Sequences.Experiments.cs

```
1  using System;
2  using LinkIndex = System.UInt64;
3  using System.Collections.Generic;
4  using Stack = System.Collections.Generic.Stack<ulong>;
5  using System.Linq;
6  using System.Text;
7  using Platform.Collections;
8  using Platform.Numbers;
9  using Platform.Data.Exceptions;
10 using Platform.Data.Sequences;
11 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
12 using Platform.Data.Doublets.Sequences.Walkers;
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     partial class Sequences
17     {
18         #region Create All Variants (Not Practical)
19
20         /// <remarks>
21         /// Number of links that is needed to generate all variants for
22         /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
23         /// </remarks>
24         public ulong[] CreateAllVariants2(ulong[] sequence)
25         {
26             return Sync.ExecuteWriteOperation(() =>
27             {
28                 if (sequence.IsNullOrEmpty())
29                 {
30                     return new ulong[0];
31                 }
32                 Links.EnsureEachLinkExists(sequence);
33                 if (sequence.Length == 1)
34                 {
35                     return sequence;
36                 }
37                 return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
38             });
39         }
40
41         private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
42         {
43             #if DEBUG
44                 if ((stopAt - startAt) < 0)
45                 {
46                     throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть меньше или
47                     ↪ 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                 var variants = new ulong[(ulong)MathHelpers.Catalan(stopAt - startAt)];
59                 var last = 0;
60                 for (var splitter = startAt; splitter < stopAt; splitter++)
61                 {
62                     var left = CreateAllVariants2Core(sequence, startAt, splitter);
63                     var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
64                     for (var i = 0; i < left.Length; i++)
65                     {
66                         for (var j = 0; j < right.Length; j++)
67                         {
68                             var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
69                             if (variant == _constants.Null)
70                             {
71                                 throw new NotImplementedException("Creation cancellation is not
72                                 ↪ implemented.");
73                             }
74                             variants[last++] = variant;
75                         }
76                     }
77                 }
78                 return variants;
79             }
80
81             public List<ulong> CreateAllVariants1(params ulong[] sequence)
82             {
83                 return Sync.ExecuteWriteOperation(() =>
84                 {
85                     if (sequence.IsNullOrEmpty())
86                     {
87                         return new List<ulong>();
88                     }
89                 });
90             }
91         }
92     }
93 }
```

```

86     }
87     Links.Unsync.EnsureEachLinkExists(sequence);
88     if (sequence.Length == 1)
89     {
90         return new List<ulong> { sequence[0] };
91     }
92     var results = new List<ulong>((int)MathHelpers.Catalan(sequence.Length));
93     return CreateAllVariants1Core(sequence, results);
94 });
95 }
96
97 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
98 {
99     if (sequence.Length == 2)
100     {
101         var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
102         if (link == _constants.Null)
103         {
104             throw new NotImplementedException("Creation cancellation is not implemented.");
105         }
106         results.Add(link);
107         return results;
108     }
109     var innerSequenceLength = sequence.Length - 1;
110     var innerSequence = new ulong[innerSequenceLength];
111     for (var li = 0; li < innerSequenceLength; li++)
112     {
113         var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
114         if (link == _constants.Null)
115         {
116             throw new NotImplementedException("Creation cancellation is not implemented.");
117         }
118         for (var isi = 0; isi < li; isi++)
119         {
120             innerSequence[isi] = sequence[isi];
121         }
122         innerSequence[li] = link;
123         for (var isi = li + 1; isi < innerSequenceLength; isi++)
124         {
125             innerSequence[isi] = sequence[isi + 1];
126         }
127         CreateAllVariants1Core(innerSequence, results);
128     }
129     return results;
130 }
131
132 #endregion
133
134 public HashSet<ulong> Each1(params ulong[] sequence)
135 {
136     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
137     Each1(link =>
138     {
139         if (!visitedLinks.Contains(link))
140         {
141             visitedLinks.Add(link); // изучить почему случаются повторы
142         }
143         return true;
144     }, sequence);
145     return visitedLinks;
146 }
147
148 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
149 {
150     if (sequence.Length == 2)
151     {
152         Links.Unsync.Each(sequence[0], sequence[1], handler);
153     }
154     else
155     {
156         var innerSequenceLength = sequence.Length - 1;
157         for (var li = 0; li < innerSequenceLength; li++)
158         {
159             var left = sequence[li];
160             var right = sequence[li + 1];
161             if (left == 0 && right == 0)
162             {
163                 continue;
164             }
165             var linkIndex = li;
166             ulong[] innerSequence = null;
167             Links.Unsync.Each(left, right, doublet =>
168             {
169                 if (innerSequence == null)
170                 {
171                     innerSequence = new ulong[innerSequenceLength];
172                     for (var isi = 0; isi < linkIndex; isi++)

```

```

173         {
174             innerSequence[isi] = sequence[isi];
175         }
176         for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
177         {
178             innerSequence[isi] = sequence[isi + 1];
179         }
180     }
181     innerSequence[linkIndex] = doublet;
182     Each1(handler, innerSequence);
183     return _constants.Continue;
184 });
185 }
186 }
187 }
188
189 public HashSet<ulong> EachPart(params ulong[] sequence)
190 {
191     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
192     EachPartCore(link =>
193     {
194         if (!visitedLinks.Contains(link))
195         {
196             visitedLinks.Add(link); // изучить почему случаются повторы
197         }
198         return true;
199     }, sequence);
200     return visitedLinks;
201 }
202
203 public void EachPart(Func<ulong, bool> handler, params ulong[] sequence)
204 {
205     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
206     EachPartCore(link =>
207     {
208         if (!visitedLinks.Contains(link))
209         {
210             visitedLinks.Add(link); // изучить почему случаются повторы
211             return handler(link);
212         }
213         return true;
214     }, sequence);
215 }
216
217 private void EachPartCore(Func<ulong, bool> handler, params ulong[] sequence)
218 {
219     if (sequence.IsNullOrEmpty())
220     {
221         return;
222     }
223     Links.EnsureEachLinkIsAnyOrExists(sequence);
224     if (sequence.Length == 1)
225     {
226         var link = sequence[0];
227         if (link > 0)
228         {
229             handler(link);
230         }
231         else
232         {
233             Links.Each(_constants.Any, _constants.Any, handler);
234         }
235     }
236     else if (sequence.Length == 2)
237     {
238         //_links.Each(sequence[0], sequence[1], handler);
239         // o_|      x_o ...
240         // x_|      |__|
241         Links.Each(sequence[1], _constants.Any, doublet =>
242         {
243             var match = Links.SearchOrDefault(sequence[0], doublet);
244             if (match != _constants.Null)
245             {
246                 handler(match);
247             }
248             return true;
249         });
250         //_x      ... x_o
251         //_o      |__|
252         Links.Each(_constants.Any, sequence[0], doublet =>
253         {
254             var match = Links.SearchOrDefault(doublet, sequence[1]);
255             if (match != 0)
256             {
257                 handler(match);
258             }
259             return true;

```

```

260     });
261     //      .-x o-.
262     //      |___|
263     PartialStepRight(x => handler(x), sequence[0], sequence[1]);
264 }
265 else
266 {
267     // TODO: Implement other variants
268     return;
269 }
270 }
271
272 private void PartialStepRight(Action<ulong> handler, ulong left, ulong right)
273 {
274     Links.Unsync.Each(_constants.Any, left, doublet =>
275     {
276         StepRight(handler, doublet, right);
277         if (left != doublet)
278         {
279             PartialStepRight(handler, doublet, right);
280         }
281         return true;
282     });
283 }
284
285 private void StepRight(Action<ulong> handler, ulong left, ulong right)
286 {
287     Links.Unsync.Each(left, _constants.Any, rightStep =>
288     {
289         TryStepRightUp(handler, right, rightStep);
290         return true;
291     });
292 }
293
294 private void TryStepRightUp(Action<ulong> handler, ulong right, ulong stepFrom)
295 {
296     var upStep = stepFrom;
297     var firstSource = Links.Unsync.GetTarget(upStep);
298     while (firstSource != right && firstSource != upStep)
299     {
300         upStep = firstSource;
301         firstSource = Links.Unsync.GetSource(upStep);
302     }
303     if (firstSource == right)
304     {
305         handler(stepFrom);
306     }
307 }
308
309 // TODO: Test
310 private void PartialStepLeft(Action<ulong> handler, ulong left, ulong right)
311 {
312     Links.Unsync.Each(right, _constants.Any, doublet =>
313     {
314         StepLeft(handler, left, doublet);
315         if (right != doublet)
316         {
317             PartialStepLeft(handler, left, doublet);
318         }
319         return true;
320     });
321 }
322
323 private void StepLeft(Action<ulong> handler, ulong left, ulong right)
324 {
325     Links.Unsync.Each(_constants.Any, right, leftStep =>
326     {
327         TryStepLeftUp(handler, left, leftStep);
328         return true;
329     });
330 }
331
332 private void TryStepLeftUp(Action<ulong> handler, ulong left, ulong stepFrom)
333 {
334     var upStep = stepFrom;
335     var firstTarget = Links.Unsync.GetSource(upStep);
336     while (firstTarget != left && firstTarget != upStep)
337     {
338         upStep = firstTarget;
339         firstTarget = Links.Unsync.GetTarget(upStep);
340     }
341     if (firstTarget == left)
342     {
343         handler(stepFrom);
344     }
345 }
346
347 private bool StartsWith(ulong sequence, ulong link)

```



```

348 {
349     var upStep = sequence;
350     var firstSource = Links.Unsync.GetSource(upStep);
351     while (firstSource != link && firstSource != upStep)
352     {
353         upStep = firstSource;
354         firstSource = Links.Unsync.GetSource(upStep);
355     }
356     return firstSource == link;
357 }
358
359 private bool EndsWith(ulong sequence, ulong link)
360 {
361     var upStep = sequence;
362     var lastTarget = Links.Unsync.GetTarget(upStep);
363     while (lastTarget != link && lastTarget != upStep)
364     {
365         upStep = lastTarget;
366         lastTarget = Links.Unsync.GetTarget(upStep);
367     }
368     return lastTarget == link;
369 }
370
371 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
372 {
373     return Sync.ExecuteReadOperation(() =>
374     {
375         var results = new List<ulong>();
376         if (sequence.Length > 0)
377         {
378             Links.EnsureEachLinkExists(sequence);
379             var firstElement = sequence[0];
380             if (sequence.Length == 1)
381             {
382                 results.Add(firstElement);
383                 return results;
384             }
385             if (sequence.Length == 2)
386             {
387                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
388                 if (doublet != _constants.Null)
389                 {
390                     results.Add(doublet);
391                 }
392                 return results;
393             }
394             var linksInSequence = new HashSet<ulong>(sequence);
395             void handler(ulong result)
396             {
397                 var filterPosition = 0;
398                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
399                     ↪ Links.Unsync.GetTarget,
400                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x, x =>
401                     {
402                         if (filterPosition == sequence.Length)
403                         {
404                             filterPosition = -2; // Длиннее чем нужно
405                             return false;
406                         }
407                         if (x != sequence[filterPosition])
408                         {
409                             filterPosition = -1;
410                             return false; // Начинается иначе
411                         }
412                         filterPosition++;
413                     }
414                     return true;
415                 });
416                 if (filterPosition == sequence.Length)
417                 {
418                     results.Add(result);
419                 }
420             }
421             if (sequence.Length >= 2)
422             {
423                 StepRight(handler, sequence[0], sequence[1]);
424             }
425             var last = sequence.Length - 2;
426             for (var i = 1; i < last; i++)
427             {
428                 PartialStepRight(handler, sequence[i], sequence[i + 1]);
429             }
430             if (sequence.Length >= 3)
431             {
432                 StepLeft(handler, sequence[sequence.Length - 2], sequence[sequence.Length - 1]);
433             }
434         }
435     });
436 }

```

```

434         return results;
435     });
436 }
437
438 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
439 {
440     return Sync.ExecuteReadOperation(() =>
441     {
442         var results = new HashSet<ulong>();
443         if (sequence.Length > 0)
444         {
445             Links.EnsureEachLinkExists(sequence);
446             var firstElement = sequence[0];
447             if (sequence.Length == 1)
448             {
449                 results.Add(firstElement);
450                 return results;
451             }
452             if (sequence.Length == 2)
453             {
454                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
455                 if (doublet != _constants.Null)
456                 {
457                     results.Add(doublet);
458                 }
459                 return results;
460             }
461             var matcher = new Matcher(this, sequence, results, null);
462             if (sequence.Length >= 2)
463             {
464                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
465             }
466             var last = sequence.Length - 2;
467             for (var i = 1; i < last; i++)
468             {
469                 PartialStepRight(matcher.AddFullMatchedToResults, sequence[i], sequence[i + 1]);
470             }
471             if (sequence.Length >= 3)
472             {
473                 StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
474                     ↪ sequence[sequence.Length - 1]);
475             }
476             return results;
477         }
478     });
479 }
480
481 public const int MaxSequenceFormatSize = 200;
482
483 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements) =>
484     ↪ FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
485
486 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
487     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
488     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
489     ↪ elementToString, insertComma, knownElements));
490
491 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
492     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params LinkIndex[]
493     ↪ knownElements)
494 {
495     var linksInSequence = new HashSet<ulong>(knownElements);
496     //var entered = new HashSet<ulong>();
497     var sb = new StringBuilder();
498     sb.Append('{');
499     if (links.Exists(sequenceLink))
500     {
501         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
502             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
503             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
504         {
505             if (insertComma && sb.Length > 1)
506             {
507                 sb.Append(',');
508             }
509             //if (entered.Contains(element))
510             //{
511                 sb.Append('{');
512                 elementToString(sb, element);
513                 sb.Append('}');
514             //}
515             //else
516             elementToString(sb, element);
517             if (sb.Length < MaxSequenceFormatSize)
518             {
519                 return true;

```

```

512         }
513         sb.Append(insertComma ? ", ..." : "...");
514         return false;
515     });
516 }
517 sb.Append('}');
518 return sb.ToString();
519 }
520
521 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements) =>
522     ↪ SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
523
524 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
525     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
526     ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync, sequenceLink,
527     ↪ elementToString, insertComma, knownElements));
528
529 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
530     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params LinkIndex[]
531     ↪ knownElements)
532 {
533     var linksInSequence = new HashSet<ulong>(knownElements);
534     var entered = new HashSet<ulong>();
535     var sb = new StringBuilder();
536     sb.Append('{');
537     if (links.Exists(sequenceLink))
538     {
539         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
540             x => linksInSequence.Contains(x) || links.IsFullPoint(x), entered.AddAndReturnVoid,
541             ↪ x => { }, entered.DoNotContains, element =>
542             {
543                 if (insertComma && sb.Length > 1)
544                 {
545                     sb.Append(',');
546                 }
547                 if (entered.Contains(element))
548                 {
549                     sb.Append('{');
550                     elementToString(sb, element);
551                     sb.Append('}');
552                 }
553                 else
554                 {
555                     elementToString(sb, element);
556                 }
557                 if (sb.Length < MaxSequenceFormatSize)
558                 {
559                     return true;
560                 }
561                 sb.Append(insertComma ? ", ..." : "...");
562                 return false;
563             });
564     }
565     sb.Append('}');
566     return sb.ToString();
567 }
568
569 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
570 {
571     return Sync.ExecuteReadOperation(() =>
572     {
573         if (sequence.Length > 0)
574         {
575             Links.EnsureEachLinkExists(sequence);
576             var results = new HashSet<ulong>();
577             for (var i = 0; i < sequence.Length; i++)
578             {
579                 AllUsagesCore(sequence[i], results);
580             }
581             var filteredResults = new List<ulong>();
582             var linksInSequence = new HashSet<ulong>(sequence);
583             foreach (var result in results)
584             {
585                 var filterPosition = -1;
586                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
587                     ↪ Links.Unsync.GetTarget,
588                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x, x =>
589                     {
590                         if (filterPosition == (sequence.Length - 1))
591                         {
592                             return false;
593                         }
594                         if (filterPosition >= 0)
595                         {
596                             if (x == sequence[filterPosition + 1])
597                             {

```

```

590         filterPosition++;
591     }
592     else
593     {
594         return false;
595     }
596 }
597 if (filterPosition < 0)
598 {
599     if (x == sequence[0])
600     {
601         filterPosition = 0;
602     }
603 }
604 return true;
605 });
606 if (filterPosition == (sequence.Length - 1))
607 {
608     filteredResults.Add(result);
609 }
610 }
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         //         AllUsagesCore(first, firstResults);
677         //         AllUsagesCore(last, lastResults);
678
679         //         firstResults.IntersectWith(lastResults);
680
681         //         //for (var i = 0; i < sequence.Length; i++)
682         //         //     AllUsagesCore(sequence[i], results);
683
684         //         var filteredResults = new HashSet<ulong>();
685         //         var matcher = new Matcher(this, sequence, filteredResults, null);
686         //         matcher.AddAllPartialMatchedToResults(firstResults);
687         //         return filteredResults;
688         //     }
689
690         //     return new HashSet<ulong>();
691     });
692 //}
693
694 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
695 {
696     return Sync.ExecuteReadOperation(() =>
697     {
698         if (sequence.Length > 0)
699         {
700             Links.EnsureEachLinkIsAnyOrExists(sequence);
701             var firstResults = new HashSet<ulong>();
702             var lastResults = new HashSet<ulong>();
703             var first = sequence.First(x => x != _constants.Any);
704             var last = sequence.Last(x => x != _constants.Any);
705             AllUsagesCore(first, firstResults);
706             AllUsagesCore(last, lastResults);
707             firstResults.IntersectWith(lastResults);
708             //for (var i = 0; i < sequence.Length; i++)
709             //     AllUsagesCore(sequence[i], results);
710             var filteredResults = new HashSet<ulong>();
711             var matcher = new Matcher(this, sequence, filteredResults, null);
712             matcher.AddAllPartialMatchedToResults(firstResults);
713             return filteredResults;
714         }
715         return new HashSet<ulong>();
716     });
717 }
718
719 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
720 ↪ IList<ulong> sequence)
721 {
722     return Sync.ExecuteReadOperation(() =>
723     {
724         if (sequence.Count > 0)
725         {
726             Links.EnsureEachLinkExists(sequence);
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             //}
744             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
745             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
746             var next = new HashSet<ulong>();
747             for (var i = 1; i < sequence.Count; i++)
748             {
749                 var collector = new AllUsagesCollector1(Links.Unsync, next);
750                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
751
752                 results.IntersectWith(next);
753                 next.Clear();
754             }
755             var filteredResults = new HashSet<ulong>();
756             var matcher = new Matcher(this, sequence, filteredResults, null, readAsElements);
757             matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x => x)); //
758             ↪ OrderBy is a Hack
759             return filteredResults;
760         }
761         return new HashSet<ulong>();
762     });
763 }

```

```

761 // Does not work
762 public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements, params
763     ↪ ulong[] sequence)
764 {
765     var visited = new HashSet<ulong>();
766     var results = new HashSet<ulong>();
767     var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return true; },
768     ↪ readAsElements);
769     var last = sequence.Length - 1;
770     for (var i = 0; i < last; i++)
771     {
772         PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
773     }
774     return results;
775 }
776 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
777 {
778     return Sync.ExecuteReadOperation(() =>
779     {
780         if (sequence.Length > 0)
781         {
782             Links.EnsureEachLinkExists(sequence);
783             //var firstElement = sequence[0];
784             //if (sequence.Length == 1)
785             //{
786                 //results.Add(firstElement);
787                 //return results;
788             //}
789             //if (sequence.Length == 2)
790             //{
791                 //var doublet = _links.SearchCore(firstElement, sequence[1]);
792                 //if (doublet != Doublets.Links.Null)
793                 //    results.Add(doublet);
794                 //return results;
795             //}
796             //var lastElement = sequence[sequence.Length - 1];
797             //Func<ulong, bool> handler = x =>
798             //{
799                 //if (StartsWith(x, firstElement) && EndsWith(x, lastElement)) results.Add(x);
800                 //return true;
801             //};
802             //if (sequence.Length >= 2)
803             //    StepRight(handler, sequence[0], sequence[1]);
804             //var last = sequence.Length - 2;
805             //for (var i = 1; i < last; i++)
806             //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
807             //if (sequence.Length >= 3)
808             //    StepLeft(handler, sequence[sequence.Length - 2], sequence[sequence.Length -
809     ↪ 1]);
810             //if (sequence.Length == 1)
811             //{
812                 //throw new NotImplementedException(); // all sequences, containing this
813     ↪ element?
814             //}
815             //if (sequence.Length == 2)
816             //{
817                 //var results = new List<ulong>();
818                 //PartialStepRight(results.Add, sequence[0], sequence[1]);
819                 //return results;
820             //}
821             //var matches = new List<List<ulong>>();
822             //var last = sequence.Length - 1;
823             //for (var i = 0; i < last; i++)
824             //{
825                 //var results = new List<ulong>();
826                 //StepRight(results.Add, sequence[i], sequence[i + 1]);
827                 //PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
828                 //if (results.Count > 0)
829                 //    matches.Add(results);
830                 //else
831                 //    return results;
832                 //if (matches.Count == 2)
833                 //{
834                     //var merged = new List<ulong>();
835                     //for (var j = 0; j < matches[0].Count; j++)
836                     //    for (var k = 0; k < matches[1].Count; k++)
837                     //        CloseInnerConnections(merged.Add, matches[0][j],
838     ↪ matches[1][k]);
839                     //if (merged.Count > 0)
840                     //    matches = new List<List<ulong>> { merged };
841                     //else
842                     //    return new List<ulong>();
843                 //}
844             //}
845             //}

```

```

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         //    sequence, 1).ToList();
861         var results = new HashSet<ulong>();
862         foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
863             firstLinkUsages, 1))
864         {
865             AllUsagesCore(match, results);
866         }
867         return results.ToList();
868     }
869     return new List<ulong>();
870 }
871
872 /// <remarks>
873 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
874 /// </remarks>
875 public HashSet<ulong> AllUsages(ulong link)
876 {
877     return Sync.ExecuteReadOperation(() =>
878     {
879         var usages = new HashSet<ulong>();
880         AllUsagesCore(link, usages);
881         return usages;
882     });
883 }
884
885 // При сборе всех использований (последовательностей) можно сохранять обратный путь к той связи
886 // с которой начинался поиск (STTTSSSTT),
887 // причём достаточно одного бита для хранения перехода влево или вправо
888 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
889 {
890     bool handler(ulong doublet)
891     {
892         if (usages.Add(doublet))
893         {
894             AllUsagesCore(doublet, usages);
895         }
896         return true;
897     }
898     Links.Unsync.Each(link, _constants.Any, handler);
899     Links.Unsync.Each(_constants.Any, link, handler);
900 }
901
902 public HashSet<ulong> AllBottomUsages(ulong link)
903 {
904     return Sync.ExecuteReadOperation(() =>
905     {
906         var visits = new HashSet<ulong>();
907         var usages = new HashSet<ulong>();
908         AllBottomUsagesCore(link, visits, usages);
909         return usages;
910     });
911 }
912
913 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong> usages)
914 {
915     bool handler(ulong doublet)
916     {
917         if (visits.Add(doublet))
918         {
919             AllBottomUsagesCore(doublet, visits, usages);
920         }
921         return true;
922     }
923     if (Links.Unsync.Count(_constants.Any, link) == 0)
924     {
925         usages.Add(link);
926     }
927 }

```

```

924     else
925     {
926         Links.Unsync.Each(link, _constants.Any, handler);
927         Links.Unsync.Each(_constants.Any, link, handler);
928     }
929 }
930
931 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
932 {
933     if (Options.UseSequenceMarker)
934     {
935         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
936             ↪ Options.MarkedSequenceMatcher, symbol);
937         return counter.Count();
938     }
939     else
940     {
941         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links, symbol);
942         return counter.Count();
943     }
944 }
945
946 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool> outerHandler)
947 {
948     bool handler(ulong doublet)
949     {
950         if (usages.Add(doublet))
951         {
952             if (!outerHandler(doublet))
953             {
954                 return false;
955             }
956             if (!AllUsagesCore1(doublet, usages, outerHandler))
957             {
958                 return false;
959             }
960         }
961         return true;
962     }
963     return Links.Unsync.Each(link, _constants.Any, handler)
964         && Links.Unsync.Each(_constants.Any, link, handler);
965 }
966
967 public void CalculateAllUsages(ulong[] totals)
968 {
969     var calculator = new AllUsagesCalculator(Links, totals);
970     calculator.Calculate();
971 }
972
973 public void CalculateAllUsages2(ulong[] totals)
974 {
975     var calculator = new AllUsagesCalculator2(Links, totals);
976     calculator.Calculate();
977 }
978
979 private class AllUsagesCalculator
980 {
981     private readonly SynchronizedLinks<ulong> _links;
982     private readonly ulong[] _totals;
983
984     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
985     {
986         _links = links;
987         _totals = totals;
988     }
989
990     public void Calculate() => _links.Each(_constants.Any, _constants.Any, 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     }

```



```

1011     }
1012 }
1013
1014 private class AllUsagesCalculator2
1015 {
1016     private readonly SynchronizedLinks<ulong> _links;
1017     private readonly ulong[] _totals;
1018
1019     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1020     {
1021         _links = links;
1022         _totals = totals;
1023     }
1024
1025     public void Calculate() => _links.Each(_constants.Any, _constants.Any, CalculateCore);
1026
1027     private bool IsElement(ulong link)
1028     {
1029         // _linksInSequence.Contains(link) ||
1030         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) == link;
1031     }
1032
1033     private bool CalculateCore(ulong link)
1034     {
1035         // TODO: Проработать защиту от заикливания
1036         // Основано на SequenceWalker.WalkLeft
1037         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1038         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1039         Func<ulong, bool> isElement = IsElement;
1040         void visitLeaf(ulong parent)
1041         {
1042             if (link != parent)
1043             {
1044                 _totals[parent]++;
1045             }
1046         }
1047         void visitNode(ulong parent)
1048         {
1049             if (link != parent)
1050             {
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                     // Обработка элемента
1074                     if (isElement(target))
1075                     {
1076                         visitLeaf(target);
1077                     }
1078                     if (isElement(source))
1079                     {
1080                         visitLeaf(source);
1081                     }
1082                     element = source;
1083                 }
1084                 else
1085                 {
1086                     stack.Push(element);
1087                     visitNode(element);
1088                     element = getTarget(element);
1089                 }
1090             }
1091             _totals[link]++;
1092             return true;
1093         }
1094     }
1095 }
1096
1097 private class AllUsagesCollector
1098 {

```

```

1099     private readonly ILinks<ulong> _links;
1100     private readonly HashSet<ulong> _usages;
1101
1102     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1103     {
1104         _links = links;
1105         _usages = usages;
1106     }
1107
1108     public bool Collect(ulong link)
1109     {
1110         if (_usages.Add(link))
1111         {
1112             _links.Each(link, _constants.Any, Collect);
1113             _links.Each(_constants.Any, link, Collect);
1114         }
1115         return true;
1116     }
1117 }
1118
1119 private class AllUsagesCollector1
1120 {
1121     private readonly ILinks<ulong> _links;
1122     private readonly HashSet<ulong> _usages;
1123     private readonly ulong _continue;
1124
1125     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1126     {
1127         _links = links;
1128         _usages = usages;
1129         _continue = _links.Constants.Continue;
1130     }
1131
1132     public ulong Collect(IList<ulong> link)
1133     {
1134         var linkIndex = _links.GetIndex(link);
1135         if (_usages.Add(linkIndex))
1136         {
1137             _links.Each(Collect, _constants.Any, linkIndex);
1138         }
1139         return _continue;
1140     }
1141 }
1142
1143 private class AllUsagesCollector2
1144 {
1145     private readonly ILinks<ulong> _links;
1146     private readonly BitString _usages;
1147
1148     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1149     {
1150         _links = links;
1151         _usages = usages;
1152     }
1153
1154     public bool Collect(ulong link)
1155     {
1156         if (_usages.Add((long)link))
1157         {
1158             _links.Each(link, _constants.Any, Collect);
1159             _links.Each(_constants.Any, link, Collect);
1160         }
1161         return true;
1162     }
1163 }
1164
1165 private class AllUsagesIntersectingCollector
1166 {
1167     private readonly SynchronizedLinks<ulong> _links;
1168     private readonly HashSet<ulong> _intersectWith;
1169     private readonly HashSet<ulong> _usages;
1170     private readonly HashSet<ulong> _enter;
1171
1172     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
1173 ↪ intersectWith, HashSet<ulong> usages)
1174     {
1175         _links = links;
1176         _intersectWith = intersectWith;
1177         _usages = usages;
1178         _enter = new HashSet<ulong>(); // защита от заикливания
1179     }
1180
1181     public bool Collect(ulong link)
1182     {
1183         if (_enter.Add(link))
1184         {
1185             if (_intersectWith.Contains(link))
1186             {
1187                 _usages.Add(link);

```

```

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, HashSet<ulong>
1223 ↪ 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, previousMatching);
1242             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage, sequence[startAt]);
1243             ↪ // почему-то эта ошибочная запись приводит к желаемым результатам.
1244             PartialStepRight(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
1245         }
1246     }
1247     if (matchings.Count == 0)
1248     {
1249         return matchings;
1250     }
1251     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1252 }
1253
1254 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong> links,
1255 ↪ params ulong[] sequence)
1256 {
1257     if (sequence == null)
1258     {
1259         return;
1260     }
1261     for (var i = 0; i < sequence.Length; i++)
1262     {
1263         if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany &&
1264             ↪ !links.Exists(sequence[i]))
1265         {
1266             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1267             ↪ $"patternSequence[{i}]");
1268         }
1269     }
1270 }
1271
1272 // Pattern Matching -> Key To Triggers

```

```

1267 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1268 {
1269     return Sync.ExecuteReadOperation(() =>
1270     {
1271         patternSequence = Simplify(patternSequence);
1272         if (patternSequence.Length > 0)
1273         {
1274             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1275             var uniqueSequenceElements = new HashSet<ulong>();
1276             for (var i = 0; i < patternSequence.Length; i++)
1277             {
1278                 if (patternSequence[i] != _constants.Any && patternSequence[i] != ZeroOrMany)
1279                 {
1280                     uniqueSequenceElements.Add(patternSequence[i]);
1281                 }
1282             }
1283             var results = new HashSet<ulong>();
1284             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1285             {
1286                 AllUsagesCore(uniqueSequenceElement, results);
1287             }
1288             var filteredResults = new HashSet<ulong>();
1289             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1290             matcher.AddAllPatternMatchedToResults(results);
1291             return filteredResults;
1292         }
1293         return new HashSet<ulong>();
1294     });
1295 }
1296
1297 // Найти все возможные связи между указанным списком связей.
1298 // Находит связи между всеми указанными связями в любом порядке.
1299 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются несколько раз в
1300 //      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             return results;
1317         }
1318     });
1319 }
1320
1321 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1322 {
1323     return Sync.ExecuteReadOperation(() =>
1324     {
1325         var results = new HashSet<ulong>();
1326         if (linksToConnect.Length > 0)
1327         {
1328             Links.EnsureEachLinkExists(linksToConnect);
1329             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1330             collector1.Collect(linksToConnect[0]);
1331             var next = new HashSet<ulong>();
1332             for (var i = 1; i < linksToConnect.Length; i++)
1333             {
1334                 var collector = new AllUsagesCollector(Links.Unsync, next);
1335                 collector.Collect(linksToConnect[i]);
1336                 results.IntersectWith(next);
1337                 next.Clear();
1338             }
1339             return results;
1340         }
1341     });
1342 }
1343
1344 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1345 {
1346     return Sync.ExecuteReadOperation(() =>
1347     {
1348         var results = new HashSet<ulong>();
1349         if (linksToConnect.Length > 0)
1350         {
1351             Links.EnsureEachLinkExists(linksToConnect);
1352             var collector1 = new AllUsagesCollector(Links, results);

```

```

1352         collector1.Collect(linksToConnect[0]);
1353         //AllUsagesCore(linksToConnect[0], results);
1354         for (var i = 1; i < linksToConnect.Length; i++)
1355         {
1356             var next = new HashSet<ulong>();
1357             var collector = new AllUsagesIntersectingCollector(Links, results, next);
1358             collector.Collect(linksToConnect[i]);
1359             //AllUsagesCore(linksToConnect[i], next);
1360             //results.IntersectWith(next);
1361             results = next;
1362         }
1363     }
1364     return results;
1365 });
1366 }
1367
1368 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1369 {
1370     return Sync.ExecuteReadOperation(() =>
1371     {
1372         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1373         ↪ BitArray((int)_links.Total + 1);
1374         if (linksToConnect.Length > 0)
1375         {
1376             Links.EnsureEachLinkExists(linksToConnect);
1377             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1378             collector1.Collect(linksToConnect[0]);
1379             for (var i = 1; i < linksToConnect.Length; i++)
1380             {
1381                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1382                 ↪ BitArray((int)_links.Total + 1);
1383                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1384                 collector.Collect(linksToConnect[i]);
1385                 results = results.And(next);
1386             }
1387             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             newLength++;
1412         }
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;
1433             }
1434             zeroOrManyStepped = true;
1435         }
1436         else
1437         {
1438             //if (zeroOrManyStepped) Is it efficient?

```

```

1437         zeroOrManyStepped = false;
1438     }
1439     newSequence[j++] = sequence[i];
1440 }
1441 return newSequence;
1442 }
1443
1444 public static void TestSimplify()
1445 {
1446     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany, ZeroOrMany,
1447     ↪ ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1448     var simplifiedSequence = Simplify(sequence);
1449 }
1450
1451 public List<ulong> GetSimilarSequences() => new List<ulong>();
1452
1453 public void Prediction()
1454 {
1455     //_links
1456     //_sequences
1457 }
1458
1459 #region From Triplets
1460
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 поддерживаются.");
1470     }
1471     var leftBound = 0;
1472     var rightBound = links.Length - 1;
1473     var left = links[leftBound++];
1474     var right = links[rightBound--];
1475     var results = new List<ulong>();
1476     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1477     return results;
1478 }
1479
1480 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[] middleLinks, ulong
1481 ↪ rightLink, int rightBound, ref List<ulong> results)
1482 {
1483     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1484     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1485     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1486     {
1487         var nextLeftLink = middleLinks[leftBound];
1488         var elements = GetRightElements(leftLink, nextLeftLink);
1489         if (leftBound <= rightBound)
1490         {
1491             for (var i = elements.Length - 1; i >= 0; i--)
1492             {
1493                 var element = elements[i];
1494                 if (element != 0)
1495                 {
1496                     CollectMatchingSequences(element, leftBound + 1, middleLinks, rightLink,
1497                     ↪ rightBound, ref results);
1498                 }
1499             }
1500         }
1501         else
1502         {
1503             for (var i = elements.Length - 1; i >= 0; i--)
1504             {
1505                 var element = elements[i];
1506                 if (element != 0)
1507                 {
1508                     results.Add(element);
1509                 }
1510             }
1511         }
1512     }
1513     else
1514     {
1515         var nextRightLink = middleLinks[rightBound];
1516         var elements = GetLeftElements(rightLink, nextRightLink);
1517         if (leftBound <= rightBound)
1518         {
1519             for (var i = elements.Length - 1; i >= 0; i--)
1520             {
1521                 var element = elements[i];
1522                 if (element != 0)
1523                 {

```

```

1521         CollectMatchingSequences(leftLink, leftBound, middleLinks, elements[i],
1522             ↪ rightBound - 1, ref results);
1523     }
1524 }
1525 else
1526 {
1527     for (var i = elements.Length - 1; i >= 0; i--)
1528     {
1529         var element = elements[i];
1530         if (element != 0)
1531         {
1532             results.Add(element);
1533         }
1534     }
1535 }
1536 }
1537 }
1538
1539 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1540 {
1541     var result = new ulong[5];
1542     TryStepRight(startLink, rightLink, result, 0);
1543     Links.Each(_constants.Any, startLink, couple =>
1544     {
1545         if (couple != startLink)
1546         {
1547             if (TryStepRight(couple, rightLink, result, 2))
1548             {
1549                 return false;
1550             }
1551             return true;
1552         });
1553     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1554     {
1555         result[4] = startLink;
1556     }
1557     return result;
1558 }
1559
1560 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1561 {
1562     var added = 0;
1563     Links.Each(startLink, _constants.Any, couple =>
1564     {
1565         if (couple != startLink)
1566         {
1567             var coupleTarget = Links.GetTarget(couple);
1568             if (coupleTarget == rightLink)
1569             {
1570                 result[offset] = couple;
1571                 if (++added == 2)
1572                 {
1573                     return false;
1574                 }
1575             }
1576             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker ==
1577                 ↪ Net.And &&
1578             {
1579                 result[offset + 1] = couple;
1580                 if (++added == 2)
1581                 {
1582                     return false;
1583                 }
1584             }
1585         }
1586         return true;
1587     });
1588     return added > 0;
1589 }
1590
1591 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1592 {
1593     var result = new ulong[5];
1594     TryStepLeft(startLink, leftLink, result, 0);
1595     Links.Each(startLink, _constants.Any, couple =>
1596     {
1597         if (couple != startLink)
1598         {
1599             if (TryStepLeft(couple, leftLink, result, 2))
1600             {
1601                 return false;
1602             }
1603         }
1604         return true;
1605     });

```

```

1606         if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1607         {
1608             result[4] = leftLink;
1609         }
1610         return result;
1611     }
1612
1613     public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1614     {
1615         var added = 0;
1616         Links.Each(_constants.Any, startLink, couple =>
1617         {
1618             if (couple != startLink)
1619             {
1620                 var coupleSource = Links.GetSource(couple);
1621                 if (coupleSource == leftLink)
1622                 {
1623                     result[offset] = couple;
1624                     if (++added == 2)
1625                     {
1626                         return false;
1627                     }
1628                 }
1629                 else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker ==
1630                     ↪ Net.And &&
1631                 {
1632                     result[offset + 1] = couple;
1633                     if (++added == 2)
1634                     {
1635                         return false;
1636                     }
1637                 }
1638             }
1639             return true;
1640         });
1641         return added > 0;
1642     }
1643
1644 #endregion
1645
1646 #region Walkers
1647
1648 public class PatternMatcher : RightSequenceWalker<ulong>
1649 {
1650     private readonly Sequences _sequences;
1651     private readonly ulong[] _patternSequence;
1652     private readonly HashSet<LinkIndex> _linksInSequence;
1653     private readonly HashSet<LinkIndex> _results;
1654
1655     #region Pattern Match
1656
1657     enum PatternBlockType
1658     {
1659         Undefined,
1660         Gap,
1661         Elements
1662     }
1663
1664     struct PatternBlock
1665     {
1666         public PatternBlockType Type;
1667         public long Start;
1668         public long Stop;
1669     }
1670
1671     private readonly List<PatternBlock> _pattern;
1672     private int _patternPosition;
1673     private long _sequencePosition;
1674
1675 #endregion
1676
1677     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence, HashSet<LinkIndex>
1678         ↪ results)
1679         : base(sequences.Links.Unsync)
1680     {
1681         _sequences = sequences;
1682         _patternSequence = patternSequence;
1683         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1684             ↪ _constants.Any && x != ZeroOrMany));
1685         _results = results;
1686         _pattern = CreateDetailedPattern();
1687     }
1688
1689     protected override bool IsElement(ICollection<ulong> link) =>
1690         ↪ _linksInSequence.Contains(Links.GetIndex(link)) || base.IsElement(link);
1691
1692     public bool PatternMatch(LinkIndex sequenceToMatch)
1693     {
1694         _patternPosition = 0;
1695     }

```



```

1691     _sequencePosition = 0;
1692     foreach (var part in Walk(sequenceToMatch))
1693     {
1694         if (!PatternMatchCore(Links.GetIndex(part)))
1695         {
1696             break;
1697         }
1698     }
1699     return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count - 1 &&
1700         ↪ _pattern[_patternPosition].Start == 0);
1701 }
1702
1703 private List<PatternBlock> CreateDetailedPattern()
1704 {
1705     var pattern = new List<PatternBlock>();
1706     var patternBlock = new PatternBlock();
1707     for (var i = 0; i < _patternSequence.Length; i++)
1708     {
1709         if (patternBlock.Type == PatternBlockType.Undefined)
1710         {
1711             if (_patternSequence[i] == _constants.Any)
1712             {
1713                 patternBlock.Type = PatternBlockType.Gap;
1714                 patternBlock.Start = 1;
1715                 patternBlock.Stop = 1;
1716             }
1717             else if (_patternSequence[i] == ZeroOrMany)
1718             {
1719                 patternBlock.Type = PatternBlockType.Gap;
1720                 patternBlock.Start = 0;
1721                 patternBlock.Stop = long.MaxValue;
1722             }
1723             else
1724             {
1725                 patternBlock.Type = PatternBlockType.Elements;
1726                 patternBlock.Start = i;
1727                 patternBlock.Stop = i;
1728             }
1729         }
1730         else if (patternBlock.Type == PatternBlockType.Elements)
1731         {
1732             if (_patternSequence[i] == _constants.Any)
1733             {
1734                 pattern.Add(patternBlock);
1735                 patternBlock = new PatternBlock
1736                 {
1737                     Type = PatternBlockType.Gap,
1738                     Start = 1,
1739                     Stop = 1
1740                 };
1741             }
1742             else if (_patternSequence[i] == ZeroOrMany)
1743             {
1744                 pattern.Add(patternBlock);
1745                 patternBlock = new PatternBlock
1746                 {
1747                     Type = PatternBlockType.Gap,
1748                     Start = 0,
1749                     Stop = long.MaxValue
1750                 };
1751             }
1752             else
1753             {
1754                 patternBlock.Stop = i;
1755             }
1756         }
1757         else // patternBlock.Type == PatternBlockType.Gap
1758         {
1759             if (_patternSequence[i] == _constants.Any)
1760             {
1761                 patternBlock.Start++;
1762                 if (patternBlock.Stop < patternBlock.Start)
1763                 {
1764                     patternBlock.Stop = patternBlock.Start;
1765                 }
1766             }
1767             else if (_patternSequence[i] == ZeroOrMany)
1768             {
1769                 patternBlock.Stop = long.MaxValue;
1770             }
1771             else
1772             {
1773                 pattern.Add(patternBlock);
1774                 patternBlock = new PatternBlock
1775                 {
1776                     Type = PatternBlockType.Elements,
1777                     Start = i,
1778                     Stop = i
1779                 };
1780             }
1781         }
1782     }
1783     pattern.Add(patternBlock);
1784     return pattern;
1785 }

```

```

1778         };
1779     }
1780 }
1781
1782 if (patternBlock.Type != PatternBlockType.Undefined)
1783 {
1784     pattern.Add(patternBlock);
1785 }
1786 return pattern;
1787 }
1788
1789 /*** match: search for regexp anywhere in text */
1790 /**int match(char* regexp, char* text)
1791 /**{
1792 /**    do
1793 /**    {
1794 /**        } while (*text++ != '\0');
1795 /**    return 0;
1796 /**}
1797
1798 /*** matchhere: search for regexp at beginning of text */
1799 /**int matchhere(char* regexp, char* text)
1800 /**{
1801 /**    if (regexp[0] == '\0')
1802 /**        return 1;
1803 /**    if (regexp[1] == '*')
1804 /**        return matchstar(regexp[0], regexp + 2, text);
1805 /**    if (regexp[0] == '$' && regexp[1] == '\0')
1806 /**        return *text == '\0';
1807 /**    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1808 /**        return matchhere(regexp + 1, text + 1);
1809 /**    return 0;
1810 /**}
1811
1812 /*** matchstar: search for c*regexp at beginning of text */
1813 /**int matchstar(int c, char* regexp, char* text)
1814 /**{
1815 /**    do
1816 /**    {
1817 /**        /* a * matches zero or more instances */
1818 /**        if (matchhere(regexp, text))
1819 /**            return 1;
1820 /**        } while (*text != '\0' && (*text++ == c || c == '.'));
1821 /**    return 0;
1822 /**}
1823
1824 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out long
1825 → maximumGap)
1826 /**{
1827 /**    mininumGap = 0;
1828 /**    maximumGap = 0;
1829 /**    element = 0;
1830 /**    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1831 /**    {
1832 /**        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1833 /**            mininumGap++;
1834 /**        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1835 /**            maximumGap = long.MaxValue;
1836 /**        else
1837 /**            break;
1838 /**    }
1839 /**    if (maximumGap < mininumGap)
1840 /**        maximumGap = mininumGap;
1841 /**}
1842
1843 private bool PatternMatchCore(LinkIndex element)
1844 {
1845     if (_patternPosition >= _pattern.Count)
1846     {
1847         _patternPosition = -2;
1848         return false;
1849     }
1850     var currentPatternBlock = _pattern[_patternPosition];
1851     if (currentPatternBlock.Type == PatternBlockType.Gap)
1852     {
1853         //var currentMatchingBlockLength = (_sequencePosition - _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

```

```

1864     }
1865     else
1866     {
1867         if (_sequencePosition > currentPatternBlock.Stop)
1868         {
1869             return false; // Соответствие невозможно
1870         }
1871         var nextPatternBlock = _pattern[_patternPosition + 1];
1872         if (_patternSequence[nextPatternBlock.Start] == element)
1873         {
1874             if (nextPatternBlock.Start < nextPatternBlock.Stop)
1875             {
1876                 _patternPosition++;
1877                 _sequencePosition = 1;
1878             }
1879             else
1880             {
1881                 _patternPosition += 2;
1882                 _sequencePosition = 0;
1883             }
1884         }
1885     }
1886 }
1887 else // currentPatternBlock.Type == PatternBlockType.Elements
1888 {
1889     var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1890     if (_patternSequence[patternElementPosition] != element)
1891     {
1892         return false; // Соответствие невозможно
1893     }
1894     if (patternElementPosition == currentPatternBlock.Stop)
1895     {
1896         _patternPosition++;
1897         _sequencePosition = 0;
1898     }
1899     else
1900     {
1901         _sequencePosition++;
1902     }
1903 }
1904 return true;
1905 //if (_patternSequence[_patternPosition] != element)
1906 //    return false;
1907 //else
1908 //{
1909 //    _sequencePosition++;
1910 //    _patternPosition++;
1911 //    return true;
1912 //}
1913 //if (_filterPosition == _patternSequence.Length)
1914 //{
1915 //    _filterPosition = -2; // Длиннее чем нужно
1916 //    return false;
1917 //}
1918 //if (element != _patternSequence[_filterPosition])
1919 //{
1920 //    _filterPosition = -1;
1921 //    return false; // Начинается иначе
1922 //}
1923 //_filterPosition++;
1924 //if (_filterPosition == (_patternSequence.Length - 1))
1925 //    return false;
1926 //if (_filterPosition >= 0)
1927 //{
1928 //    if (element == _patternSequence[_filterPosition + 1])
1929 //        _filterPosition++;
1930 //    else
1931 //        return false;
1932 //}
1933 //if (_filterPosition < 0)
1934 //{
1935 //    if (element == _patternSequence[0])
1936 //        _filterPosition = 0;
1937 //}
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         /// определяющее 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 = !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i -
33                 ↪ 1], sequence[i]), _null))) { }
34             for (; i >= 1; i--)
35             {

```

```

34         _links.GetOrCreate(sequence[i - 1], sequence[i]);
35     }
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], sequence[i]),
48             ↪ _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 = !_equalityComparer.Equals(links.SearchOrDefault(sequence[i -
69     ↪ 1], sequence[i]), _null))) { }
70     for (; i >= 1; i--)
71     {
72         links.GetOrCreate(sequence[i - 1], sequence[i]);
73     }
74     return indexed;
75 }
76
77 public bool CheckIndex(IList<TLink> sequence)
78 {
79     var indexed = true;
80     var i = sequence.Count;
81     while (--i >= 1 && (indexed = !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i -
82     ↪ 1], sequence[i]), _null))) { }
83     return indexed;
84 }
85 }
86 }

```

## ./Sequences/SequencesOptions.cs

```

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

```

```

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
97 public void ValidateOptions()
98 {
99     if (UseGarbageCollection && !UseSequenceMarker)
100     {
101         throw new NotSupportedException("To use garbage collection UseSequenceMarker option
102                                     ↪ must be on.");
103     }
104 }
105 }
106 }
107 }

```

## ./Sequences/UnicodeMap.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Globalization;
4 using System.Runtime.CompilerServices;
5 using System.Text;
6 using Platform.Data.Sequences;
7
8 namespace Platform.Data.Doublets.Sequences
9 {
10     public class UnicodeMap

```

```

11 {
12     public static readonly ulong FirstCharLink = 1;
13     public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
14     public static readonly ulong MapSize = 1 + char.MaxValue;
15
16     private readonly ILinks<ulong> _links;
17     private bool _initialized;
18
19     public UnicodeMap(ILinks<ulong> links) => _links = links;
20
21     public static UnicodeMap InitNew(ILinks<ulong> links)
22     {
23         var map = new UnicodeMap(links);
24         map.Init();
25         return map;
26     }
27
28     public void Init()
29     {
30         if (_initialized)
31         {
32             return;
33         }
34         _initialized = true;
35         var firstLink = _links.CreatePoint();
36         if (firstLink != FirstCharLink)
37         {
38             _links.Delete(firstLink);
39         }
40         else
41         {
42             for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
43             {
44                 // From NIL to It (NIL -> Character) transformation meaning, (or infinite amount of
45                 // ↪ NIL characters before actual Character)
46                 var createdLink = _links.CreatePoint();
47                 _links.Update(createdLink, firstLink, createdLink);
48                 if (createdLink != i)
49                 {
50                     throw new InvalidOperationException("Unable to initialize UTF 16 table.");
51                 }
52             }
53         }
54
55         // 0 - null link
56         // 1 - nil character (0 character)
57         // ...
58         // 65536 (0(1) + 65535 = 65536 possible values)
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public static ulong FromCharToLink(char character) => (ulong)character + 1;
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         public static char FromLinkToChar(ulong link) => (char)(link - 1);
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         public static bool IsCharLink(ulong link) => link <= MapSize;
68
69         public static string FromLinksToString(IList<ulong> linksList)
70         {
71             var sb = new StringBuilder();
72             for (int i = 0; i < linksList.Count; i++)
73             {
74                 sb.Append(FromLinkToChar(linksList[i]));
75             }
76             return sb.ToString();
77         }
78
79         public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
80         {
81             var sb = new StringBuilder();
82             if (links.Exists(link))
83             {
84                 StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
85                     x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x, element =>
86                 {
87                     sb.Append(FromLinkToChar(element));
88                     return true;
89                 });
90             }
91             return sb.ToString();
92         }
93
94         public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
95             ↪ chars.Length);
96
97         public static ulong[] FromCharsToLinkArray(char[] chars, int count)

```



```

97     {
98         // char array to ulong array
99         var linksSequence = new ulong[count];
100         for (var i = 0; i < count; i++)
101         {
102             linksSequence[i] = FromCharToLink(chars[i]);
103         }
104         return linksSequence;
105     }
106
107     public static ulong[] FromStringToLinkArray(string sequence)
108     {
109         // char array to ulong array
110         var linksSequence = new ulong[sequence.Length];
111         for (var i = 0; i < sequence.Length; i++)
112         {
113             linksSequence[i] = FromCharToLink(sequence[i]);
114         }
115         return linksSequence;
116     }
117
118     public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
119     {
120         var result = new List<ulong[]>();
121         var offset = 0;
122         while (offset < sequence.Length)
123         {
124             var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
125             var relativeLength = 1;
126             var absoluteLength = offset + relativeLength;
127             while (absoluteLength < sequence.Length &&
128                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
129             {
130                 relativeLength++;
131                 absoluteLength++;
132             }
133             // char array to ulong array
134             var innerSequence = new ulong[relativeLength];
135             var maxLength = offset + relativeLength;
136             for (var i = offset; i < maxLength; i++)
137             {
138                 innerSequence[i - offset] = FromCharToLink(sequence[i]);
139             }
140             result.Add(innerSequence);
141             offset += relativeLength;
142         }
143         return result;
144     }
145
146     public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
147     {
148         var result = new List<ulong[]>();
149         var offset = 0;
150         while (offset < array.Length)
151         {
152             var relativeLength = 1;
153             if (array[offset] <= LastCharLink)
154             {
155                 var currentCategory =
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(array[a
161                        bsoluteLength])))
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             }
176             // copy array
177             var innerSequence = new ulong[relativeLength];
178             var maxLength = offset + relativeLength;
179             for (var i = offset; i < maxLength; i++)
180             {
181                 innerSequence[i - offset] = array[i];
182             }
183             result.Add(innerSequence);
184             offset += relativeLength;
185         }
186     }

```

```

183     }
184     return result;
185 }
186 }
187 }

```

#### ./Sequences/Walkers/LeftSequenceWalker.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 namespace Platform.Data.Doublets.Sequences.Walkers
5 {
6     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
7     {
8         public LeftSequenceWalker(ILinks<TLink> links) : base(links) { }
9
10        [MethodImpl(MethodImplOptions.AggressiveInlining)]
11        protected override IList<TLink> GetNextElementAfterPop(IList<TLink> element) =>
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>, ISequenceWalker<TLink>
8     {
9         // TODO: Use IStack instead of System.Collections.Generic.Stack, but IStack should contain
10         ↪ IsEmpty property
11         private readonly Stack<IList<TLink>> _stack;
12
13         protected SequenceWalkerBase(ILinks<TLink> links) : base(links) => _stack = new
14             ↪ Stack<IList<TLink>>();
15     }
16 }

```

```

13 public IEnumerable<IList<TLink>> Walk(TLink sequence)
14 {
15     if (_stack.Count > 0)
16     {
17         _stack.Clear(); // This can be replaced with while(!_stack.IsEmpty) _stack.Pop()
18     }
19     var element = Links.GetLink(sequence);
20     if (IsElement(element))
21     {
22         yield return element;
23     }
24     else
25     {
26         while (true)
27         {
28             if (IsElement(element))
29             {
30                 if (_stack.Count == 0)
31                 {
32                     break;
33                 }
34                 element = _stack.Pop();
35                 foreach (var output in WalkContents(element))
36                 {
37                     yield return output;
38                 }
39                 element = GetNextElementAfterPop(element);
40             }
41             else
42             {
43                 _stack.Push(element);
44                 element = GetNextElementAfterPush(element);
45             }
46         }
47     }
48 }
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected virtual bool IsElement(IList<TLink> elementLink) =>
52     ⇨ Point<TLink>.IsPartialPointUnchecked(elementLink);
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected abstract IList<TLink> GetNextElementAfterPop(IList<TLink> element);
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected abstract IList<TLink> GetNextElementAfterPush(IList<TLink> element);
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected abstract IEnumerable<IList<TLink>> WalkContents(IList<TLink> element);
62 }
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         }
37     }
38 }

```

```

34     }
35     return element;
36 }
37
38 public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
    ↪ _links.GetOrCreate(GetTop(), element));
39 }
40 }

```

#### ./Stacks/StackExtensions.cs

```

1 namespace Platform.Data.Doublets.Stacks
2 {
3     public static class StackExtensions
4     {
5         public static TLink CreateStack<TLink>(this ILinks<TLink> 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(), links)
22            ↪ { }
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, IList<T>
43        ↪ substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
44        //{
45        //    if (restriction != null && substitution != null && !substitution.EqualTo(restriction))
46        //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler, substitution,
47        ↪ substitutedHandler, Unsync.Trigger);
48        //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
49        ↪ substitutedHandler, Unsync.Trigger);
50    }
51 }

```

#### ./UInt64Link.cs

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using Platform.Exceptions;
5 using Platform.Ranges;

```

```

6  using Platform.Helpers.Singletons;
7  using Platform.Data.Constants;
8
9  namespace Platform.Data.Doublets
10 {
11     /// <summary>
12     /// Структура описывающая уникальную связь.
13     /// </summary>
14     public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
15     {
16         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
17             ↳ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
18
19         private const int Length = 3;
20
21         public readonly ulong Index;
22         public readonly ulong Source;
23         public readonly ulong Target;
24
25         public static readonly UInt64Link Null = new UInt64Link();
26
27         public UInt64Link(params ulong[] values)
28         {
29             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
30                 ↳ _constants.Null;
31             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
32                 ↳ _constants.Null;
33             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
34                 ↳ _constants.Null;
35         }
36
37         public UInt64Link(IList<ulong> values)
38         {
39             Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
40                 ↳ _constants.Null;
41             Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
42                 ↳ _constants.Null;
43             Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
44                 ↳ _constants.Null;
45         }
46
47         public UInt64Link(ulong index, ulong source, ulong target)
48         {
49             Index = index;
50             Source = source;
51             Target = target;
52         }
53
54         public UInt64Link(ulong source, ulong target)
55             : this(_constants.Null, source, target)
56         {
57             Source = source;
58             Target = target;
59         }
60
61         public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source, 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 && Equals((UInt64Link)other);
70
71         public bool Equals(UInt64Link other) => Index == other.Index
72             && Source == other.Source
73             && Target == other.Target;
74
75         public static string ToString(ulong index, ulong source, ulong target) => $"{index}:
76             ↳ {source}->{target}";
77
78         public static string ToString(ulong source, ulong target) => $"{source}->{target}";
79
80         public static implicit operator ulong[] (UInt64Link link) => link.ToArray();
81
82         public static implicit operator UInt64Link(ulong[] linkArray) => new UInt64Link(linkArray);
83
84         public ulong[] ToArray()
85         {
86             var array = new ulong[Length];
87             CopyTo(array, 0);
88             return array;
89         }
90
91         public override string ToString() => Index == _constants.Null ? ToString(Source, Target) :
92             ↳ ToString(Index, Source, Target);
93
94         #region IList

```

```

86
87 public ulong this[int index]
88 {
89     get
90     {
91         Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1), nameof(index));
92         if (index == _constants.IndexPart)
93         {
94             return Index;
95         }
96         if (index == _constants.SourcePart)
97         {
98             return Source;
99         }
100        if (index == _constants.TargetPart)
101        {
102            return Target;
103        }
104        throw new NotSupportedException(); // Impossible path due to Ensure.ArgumentInRange
105    }
106    set => throw new NotSupportedException();
107 }
108
109 public int Count => Length;
110
111 public bool IsReadOnly => true;
112
113 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
114
115 public IEnumerator<ulong> GetEnumerator()
116 {
117     yield return Index;
118     yield return Source;
119     yield return Target;
120 }
121
122 public void Add(ulong item) => throw new NotSupportedException();
123
124 public void Clear() => throw new NotSupportedException();
125
126 public bool Contains(ulong item) => IndexOf(item) >= 0;
127
128 public void CopyTo(ulong[] array, int arrayIndex)
129 {
130     Ensure.Always.ArgumentNotNull(array, nameof(array));
131     Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
132         ↪   nameof(arrayIndex));
133     if (arrayIndex + Length > array.Length)
134     {
135         throw new ArgumentException();
136     }
137     array[arrayIndex++] = Index;
138     array[arrayIndex++] = Source;
139     array[arrayIndex] = Target;
140 }
141
142 public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
143
144 public int IndexOf(ulong item)
145 {
146     if (Index == item)
147     {
148         return _constants.IndexPart;
149     }
150     if (Source == item)
151     {
152         return _constants.SourcePart;
153     }
154     if (Target == item)
155     {
156         return _constants.TargetPart;
157     }
158     return -1;
159 }
160
161 public void Insert(int index, ulong item) => throw new NotSupportedException();
162
163 public void RemoveAt(int index) => throw new NotSupportedException();
164
165 #endregion
166 }
167

```

./UInt64LinkExtensions.cs

```

1 namespace Platform.Data.Doublets
2 {
3     public static class UInt64LinkExtensions
4     {

```

```

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

```

# ./UInt64LinksExtensions.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using Platform.Helpers.Singletons;
5  using Platform.Data.Constants;
6  using Platform.Data.Exceptions;
7  using Platform.Data.Doublets.Sequences;
8
9  namespace Platform.Data.Doublets
10 {
11     public static class UInt64LinksExtensions
12     {
13         public static readonly LinksCombinedConstants<bool, ulong, int> 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], $"sequence[{i}]");
29                 }
30             }
31
32         public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong> sequence)
33         {
34             if (sequence == null)
35             {
36                 return;
37             }
38             for (var i = 0; i < sequence.Count; i++)
39             {
40                 if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
41                 {
42                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i], $"sequence[{i}]");
43                 }
44             }
45
46         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
47         {
48             if (sequence == null)
49             {
50                 return false;
51             }
52             var constants = links.Constants;
53             for (var i = 0; i < sequence.Length; i++)
54             {
55                 if (sequence[i] == constants.Any)
56                 {
57                     return true;
58                 }
59             }
60             return false;
61         }
62
63         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
64             ↳ 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) =>
69                 ↳ innerSb.Append(link.Index), renderIndex, renderDebug);
70             return sb.ToString();
71
72         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
73             ↳ Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement, bool
74             ↳ renderIndex = false, bool renderDebug = false)
75         {
76             var sb = new StringBuilder();
77             var visited = new HashSet<ulong>();
78             links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
79                 ↳ 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, appendElement,
    ↪ renderIndex);
132                     }
133                 }
134                 sb.Append(')');
135             }
136             else
137             {
138                 if (renderDebug)
139                 {
140                     sb.Append('*');
141                 }
142                 sb.Append(linkIndex);
143             }
144         }
145         else
146         {
147             if (renderDebug)
148             {
149                 sb.Append('~');
150             }
151             sb.Append(linkIndex);
152         }
153     }
154 }
155 }

```



./UInt64LinksTransactionsLayer.cs

```
1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Runtime.CompilerServices;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Platform.Disposables;
9  using Platform.Timestamps;
10 using Platform.Unsafe;
11 using Platform.IO;
12 using Platform.Data.Doublets.Decorators;
13
14 namespace Platform.Data.Doublets
15 {
16     public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
17     {
18         /// <remarks>
19         /// Альтернативные варианты хранения трансформации (элемента транзакции):
20         ///
21         /// private enum TransitionType
22         /// {
23         ///     Creation,
24         ///     UpdateOf,
25         ///     UpdateTo,
26         ///     Deletion
27         /// }
28         ///
29         /// private struct Transition
30         /// {
31         ///     public ulong TransactionId;
32         ///     public UniqueTimestamp Timestamp;
33         ///     public TransactionItemType Type;
34         ///     public Link Source;
35         ///     public Link Linker;
36         ///     public Link Target;
37         /// }
38         ///
39         /// Или
40         ///
41         /// public struct TransitionHeader
42         /// {
43         ///     public ulong TransactionIdCombined;
44         ///     public ulong TimestampCombined;
45         ///
46         ///     public ulong TransactionId
47         ///     {
48         ///         get
49         ///         {
50             ///         return (ulong) mask & TransactionIdCombined;
51         ///     }
52         /// }
53         ///
54         ///     public UniqueTimestamp Timestamp
55         ///     {
56         ///         get
57         ///         {
58             ///         return (UniqueTimestamp)mask & TransactionIdCombined;
59         ///     }
60         /// }
61         ///
62         ///     public TransactionItemType Type
63         ///     {
64         ///         get
65         ///         {
66             ///         // Использовать по одному биту из TransactionId и Timestamp,
67             ///         // для значения в 2 бита, которое представляет тип операции
68             ///         throw new NotImplementedException();
69         ///     }
70         /// }
71         /// }
72         ///
73         /// private struct Transition
74         /// {
75         ///     public TransitionHeader Header;
76         ///     public Link Source;
77         ///     public Link Linker;
78         ///     public Link Target;
79         /// }
80         ///
81         /// </remarks>
82         public struct Transition
83         {
84             public static readonly long Size = StructureHelpers.SizeOf<Transition>();
85
86             public readonly ulong TransactionId;
```

```

87     public readonly UInt64Link Before;
88     public readonly UInt64Link After;
89     public readonly Timestamp Timestamp;
90
91     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId,
92         ↪ UInt64Link before, UInt64Link after)
93     {
94         TransactionId = transactionId;
95         Before = before;
96         After = after;
97         Timestamp = uniqueTimestampFactory.Create();
98     }
99
100    public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId,
101        ↪ UInt64Link before)
102        : this(uniqueTimestampFactory, transactionId, before, default)
103    {
104    }
105
106    public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
107        : this(uniqueTimestampFactory, transactionId, default, default)
108    {
109    }
110
111    public override string ToString() => $"{Timestamp} {TransactionId}: {Before} => {After}";
112 }
113
114 /// <remarks>
115 /// Другие варианты реализации транзакций (атомарности):
116 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker Target)) и
117 ///    ↪ индексов.
118 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
119 ///    ↪ потребуется решить вопрос
120 ///    со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
121 ///    ↪ пересечениями идентификаторов.
122 ///
123 /// Где хранить промежуточный список транзакций?
124 ///
125 /// В оперативной памяти:
126 /// Минусы:
127 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
128 ///    так как нужно отдельно выделять память под список трансформаций.
129 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
130 ///    если транзакция использует слишком много трансформаций.
131 ///    -> Можно использовать жёсткий диск для слишком длинных транзакций.
132 ///    -> Максимальный размер списка трансформаций можно ограничить / задать константой.
133 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом создавая
134 ///    ↪ задержку.
135 ///
136 /// На жёстком диске:
137 /// Минусы:
138 /// 1. Длительный отклик, на запись каждой трансформации.
139 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
140 ///    -> Это может решаться упаковкой/исключением дублирующих операций.
141 ///    -> Также это может решаться тем, что короткие транзакции вообще
142 ///        не будут записываться в случае отката.
143 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все операции
144 ///    ↪ (трансформации)
145 ///        будут записаны в лог.
146 ///
147 /// </remarks>
148 public class Transaction : DisposableBase
149 {
150     private readonly Queue<Transition> _transitions;
151     private readonly UInt64LinksTransactionsLayer _layer;
152     public bool IsCommitted { get; private set; }
153     public bool IsReverted { get; private set; }
154
155     public Transaction(UInt64LinksTransactionsLayer layer)
156     {
157         _layer = layer;
158         if (_layer._currentTransactionId != 0)
159         {
160             throw new NotSupportedException("Nested transactions not supported.");
161         }
162         IsCommitted = false;
163         IsReverted = false;
164         _transitions = new Queue<Transition>();
165         SetCurrentTransaction(layer, this);
166     }
167
168     public void Commit()
169     {
170         EnsureTransactionAllowsWriteOperations(this);
171         while (_transitions.Count > 0)
172         {
173             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
↵ https://github.com/linksplatform/Disposables/issues/13 FIXED
217 protected override bool AllowMultipleDisposeCalls => true;
218
219
220 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
221
222 private readonly string _logAddress;
223 private readonly FileStream _log;
224 private readonly Queue<Transition> _transitions;
225 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
226 private Task _transitionsPusher;
227 private Transition _lastCommittedTransition;
228 private ulong _currentTransactionId;
229 private Queue<Transition> _currentTransactionTransitions;
230 private Transaction _currentTransaction;
231 private ulong _lastCommittedTransactionId;
232
233 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
234 : base(links)
235 {
236     if (string.IsNullOrEmpty(logAddress))
237     {
238         throw new ArgumentNullException(nameof(logAddress));
239     }
240     // В первой строке файла хранится последняя закоммиченную транзакцию.
241     // При запуске это используется для проверки удачного закрытия файла лога.
242     // In the first line of the file the last committed transaction is stored.
243     // On startup, this is used to check that the log file is successfully closed.
244     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
245     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
246     if (!lastCommittedTransition.Equals(lastWrittenTransition))
247     {
248         Dispose();
249         throw new NotSupportedException("Database is damaged, autorecovery is not supported
↵ yet.");
250     }
251     if (lastCommittedTransition.Equals(default(Transition)))

```

```

252     {
253         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
254     }
255     _lastCommittedTransition = lastCommittedTransition;
256     // TODO: Think about a better way to calculate or store this value
257     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
258     _lastCommittedTransactionId = allTransitions.Max(x => x.TransactionId);
259     _uniqueTimestampFactory = new UniqueTimestampFactory();
260     _logAddress = logAddress;
261     _log = FileHelpers.Append(logAddress);
262     _transitions = new Queue<Transition>();
263     _transitionsPusher = new Task(TransitionsPusher);
264     _transitionsPusher.Start();
265 }
266
267 public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
268
269 public override ulong Create()
270 {
271     var createdLinkIndex = Links.Create();
272     var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
273     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId, default,
274         ↪ createdLink));
275     return createdLinkIndex;
276 }
277
278 public override ulong Update(IList<ulong> parts)
279 {
280     var beforeLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
281     parts[Constants.IndexPart] = Links.Update(parts);
282     var afterLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
283     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId, beforeLink,
284         ↪ afterLink));
285     return parts[Constants.IndexPart];
286 }
287
288 public override void Delete(ulong link)
289 {
290     var deletedLink = new UInt64Link(Links.GetLink(link));
291     Links.Delete(link);
292     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
293         ↪ deletedLink, default));
294 }
295
296 [MethodImpl(MethodImplOptions.AggressiveInlining)]
297 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
298     ↪ _transitions;
299
300 private void CommitTransition(Transition transition)
301 {
302     if (_currentTransaction != null)
303     {
304         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
305     }
306     var transitions = GetCurrentTransitions();
307     transitions.Enqueue(transition);
308 }
309
310 private void RevertTransition(Transition transition)
311 {
312     if (transition.After.IsNull()) // Revert Deletion with Creation
313     {
314         Links.Create();
315     }
316     else if (transition.Before.IsNull()) // Revert Creation with Deletion
317     {
318         Links.Delete(transition.After.Index);
319     }
320     else // Revert Update
321     {
322         Links.Update(new[] { transition.After.Index, transition.Before.Source,
323             ↪ transition.Before.Target });
324     }
325 }
326
327 private void ResetCurrentTransation()
328 {
329     _currentTransactionId = 0;
330     _currentTransactionTransitions = null;
331     _currentTransaction = null;
332 }
333
334 private void PushTransitions()
335 {
336     if (_log == null || _transitions == null)
337     {
338         return;
339     }

```

```

334     }
335     for (var i = 0; i < _transitions.Count; i++)
336     {
337         var transition = _transitions.Dequeue();
338
339         _log.Write(transition);
340         _lastCommittedTransition = transition;
341     }
342 }
343
344 private void TransitionsPusher()
345 {
346     while (!IsDisposed && _transitionsPusher != null)
347     {
348         Thread.Sleep(DefaultPushDelay);
349         PushTransitions();
350     }
351 }
352
353 public Transaction BeginTransaction() => new Transaction(this);
354
355 private void DisposeTransitions()
356 {
357     try
358     {
359         var pusher = _transitionsPusher;
360         if (pusher != null)
361         {
362             _transitionsPusher = null;
363             pusher.Wait();
364         }
365         if (_transitions != null)
366         {
367             PushTransitions();
368         }
369         Disposable.TryDispose(_log);
370         FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
371     }
372     catch
373     {
374     }
375 }
376
377 #region DisposalBase
378
379 protected override void DisposeCore(bool manual, bool wasDisposed)
380 {
381     if (!wasDisposed)
382     {
383         DisposeTransitions();
384     }
385     base.DisposeCore(manual, wasDisposed);
386 }
387
388 #endregion
389 }
390 }

```

## Index

- ./Converters/AddressToUnaryNumberConverter.cs, 1
- ./Converters/LinkToltsFrequencyNumberConveter.cs, 1
- ./Converters/PowerOf2ToUnaryNumberConverter.cs, 2
- ./Converters/UnaryNumberToAddressAddOperationConverter.cs, 2
- ./Converters/UnaryNumberToAddressOrOperationConverter.cs, 3
- ./Decorators/LinksCascadeDependenciesResolver.cs, 3
- ./Decorators/LinksCascadeUniquenessAndDependenciesResolver.cs, 4
- ./Decorators/LinksDecoratorBase.cs, 4
- ./Decorators/LinksDependenciesValidator.cs, 5
- ./Decorators/LinksDisposableDecoratorBase.cs, 5
- ./Decorators/LinksInnerReferenceValidator.cs, 5
- ./Decorators/LinksNonExistentReferencesCreator.cs, 6
- ./Decorators/LinksNullToSelfReferenceResolver.cs, 6
- ./Decorators/LinksSelfReferenceResolver.cs, 7
- ./Decorators/LinksUniquenessResolver.cs, 7
- ./Decorators/LinksUniquenessValidator.cs, 7
- ./Decorators/NonNullContentsLinkDeletionResolver.cs, 8
- ./Decorators/UInt64Links.cs, 8
- ./Decorators/UniLinks.cs, 9
- ./Doublet.cs, 14
- ./DoubletComparer.cs, 13
- ./Hybrid.cs, 14
- ./ILinks.cs, 15
- ./ILinksExtensions.cs, 15
- ./ISynchronizedLinks.cs, 24
- ./Incrementers/FrequencyIncrementer.cs, 23
- ./Incrementers/LinkFrequencyIncrementer.cs, 23
- ./Incrementers/UnaryNumberIncrementer.cs, 24
- ./Link.cs, 24
- ./LinkExtensions.cs, 26
- ./LinksOperatorBase.cs, 26
- ./PropertyOperators/DefaultLinkPropertyOperator.cs, 27
- ./PropertyOperators/FrequencyPropertyOperator.cs, 27
- ./ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs, 36
- ./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs, 36
- ./ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 28
- ./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs, 48
- ./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs, 48
- ./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 42
- ./Sequences/Converters/BalancedVariantConverter.cs, 54
- ./Sequences/Converters/CompressingConverter.cs, 54
- ./Sequences/Converters/LinksListToSequenceConverterBase.cs, 57
- ./Sequences/Converters/OptimalVariantConverter.cs, 57
- ./Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 59
- ./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs, 59
- ./Sequences/CreteriaMatchers/MarkedSequenceCreteriaMatcher.cs, 59
- ./Sequences/DefaultSequenceAppender.cs, 60
- ./Sequences/DuplicateSegmentsCounter.cs, 60
- ./Sequences/DuplicateSegmentsProvider.cs, 60
- ./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs, 62
- ./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToltsFrequencyNumberConverter.cs, 62
- ./Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 63
- ./Sequences/Frequencies/Cache/LinkFrequency.cs, 64
- ./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 65
- ./Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 65
- ./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 65
- ./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 66
- ./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 66
- ./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 66
- ./Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 67
- ./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 68
- ./Sequences/HeightProviders/ISequenceHeightProvider.cs, 68
- ./Sequences/Sequences.Experiments.ReadSequence.cs, 100
- ./Sequences/Sequences.Experiments.cs, 77
- ./Sequences/Sequences.cs, 68
- ./Sequences/SequencesExtensions.cs, 101
- ./Sequences/SequencesIndexer.cs, 101
- ./Sequences/SequencesOptions.cs, 102
- ./Sequences/UnicodeMap.cs, 103
- ./Sequences/Walkers/LeftSequenceWalker.cs, 106
- ./Sequences/Walkers/RightSequenceWalker.cs, 106
- ./Sequences/Walkers/SequenceWalkerBase.cs, 106

./Stacks/Stack.cs, 107  
./Stacks/StackExtensions.cs, 108  
./SynchronizedLinks.cs, 108  
./UInt64Link.cs, 108  
./UInt64LinkExtensions.cs, 110  
./UInt64LinksExtensions.cs, 111  
./UInt64LinksTransactionsLayer.cs, 112  
./obj/Debug/netstandard2.0/Platform.Data.Doublets.AssemblyInfo.cs, 27