

LinksPlatform's Platform.Data.Doublets Class Library

./Converters/AddressToUnaryNumberConverter.cs

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

./Converters/LinkToItsFrequencyNumberConveter.cs

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

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

./Converters/PowerOf2ToUnaryNumberConverter.cs

```
1  \texttt{
2  using System;
3  using System.Collections.Generic;
4  using Platform.Interfaces;
5
6  namespace Platform.Data.Doublets.Converters
7  {
8      public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
9         ↪ IConverter<int, TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly TLink[] _unaryNumberPowersOf2;
15
16         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) :
17             ↪ base(links)
18         {
19             _unaryNumberPowersOf2 = new TLink[64];
20             _unaryNumberPowersOf2[0] = one;
21         }
22
23         public TLink Convert(int power)
```

```

21     {
22         if (power < 0 || power >= _unaryNumberPowersOf2.Length)
23         {
24             throw new ArgumentOutOfRangeException(nameof(power));
25         }
26         if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
27         {
28             return _unaryNumberPowersOf2[power];
29         }
30         var previousPowerOf2 = Convert(power - 1);
31         var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
32         _unaryNumberPowersOf2[power] = powerOf2;
33         return powerOf2;
34     }
35 }
36 }
37 }

```

./Converters/UnaryNumberToAddressAddOperationConverter.cs

```

1  [texttt{
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Numbers;
5
6  namespace Platform.Data.Doublets.Converters
7  {
8      public class UnaryNumberToAddressAddOperationConverter<TLink> :
9          ↳ LinksOperatorBase<TLink>, IConverter<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↳ EqualityComparer<TLink>.Default;
13
14         private Dictionary<TLink, TLink> _unaryToUInt64;
15         private readonly TLink _unaryOne;
16
17         public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink
18             ↳ unaryOne)
19             : base(links)
20         {
21             _unaryOne = unaryOne;
22             InitUnaryToUInt64();
23         }
24
25         private void InitUnaryToUInt64()
26         {
27             _unaryToUInt64 = new Dictionary<TLink, TLink>
28             {
29                 { _unaryOne, Integer<TLink>.One }
30             };
31             var unary = _unaryOne;
32             var number = Integer<TLink>.One;
33             for (var i = 1; i < 64; i++)
34             {
35                 _unaryToUInt64.Add(unary = Links.GetOrCreate(unary, unary), number =
36                     ↳ (Integer<TLink>)((Integer<TLink>)number * 2UL));
37             }
38         }
39
40         public TLink Convert(TLink unaryNumber)
41         {
42             if (!_equalityComparer.Equals(unaryNumber, default))
43             {
44                 return default;
45             }
46         }
47     }
48 }

```

```

41     }
42     if (!_equalityComparer.Equals(unaryNumber, _unaryOne))
43     {
44         return Integer<TLink>.One;
45     }
46     var source = Links.GetSource(unaryNumber);
47     var target = Links.GetTarget(unaryNumber);
48     if (!_equalityComparer.Equals(source, target))
49     {
50         return _unaryToUInt64[unaryNumber];
51     }
52     else
53     {
54         var result = _unaryToUInt64[source];
55         TLink lastValue;
56         while (!_unaryToUInt64.TryGetValue(target, out lastValue))
57         {
58             source = Links.GetSource(target);
59             result = ArithmeticHelpers.Add(result, _unaryToUInt64[source]);
60             target = Links.GetTarget(target);
61         }
62         result = ArithmeticHelpers.Add(result, lastValue);
63         return result;
64     }
65 }
66 }
67 }
68 }

```

./Converters/UnaryNumberToAddressOrOperationConverter.cs

```

1  [texttt{
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Reflection;
5  using Platform.Numbers;
6
7  namespace Platform.Data.Doublets.Converters
8  {
9      public class UnaryNumberToAddressOrOperationConverter<TLink> :
10          ↳ LinksOperatorBase<TLink>, IConverter<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↳ EqualityComparer<TLink>.Default;
14
15         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
16
17         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links,
18             ↳ IConverter<int, TLink> powerOf2ToUnaryNumberConverter)
19             : base(links)
20         {
21             _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
22             for (int i = 0; i < CachedTypeInfo<TLink>.BitsLength; i++)
23             {
24                 _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Con
25                     ↳ vert(i),
26                     ↳ i);
27             }
28         }
29
30         public TLink Convert(TLink sourceNumber)
31         {
32         }
33     }
34 }

```

```

27     var source = sourceNumber;
28     var target = Links.Constants.Null;
29     while (!_equalityComparer.Equals(source, Links.Constants.Null))
30     {
31         if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int
            ↪ powerOf2Index))
32         {
33             source = Links.Constants.Null;
34         }
35         else
36         {
37             powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
38             source = Links.GetTarget(source);
39         }
40         target = (Integer<TLink>)((Integer<TLink>)target | 1UL << powerOf2Index);
            ↪ // MathHelpers.Or(target, MathHelpers.ShiftLeft(One, powerOf2Index))
41     }
42     return target;
43 }
44 }
45 }
46 }

```

./Decorators/LinksCascadeDependenciesResolver.cs

```

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

```

```

36     }
37 }
38 }
39 }

```

./Decorators/LinksCascadeUniquenessAndDependenciesResolver.cs

```

1  namespace Platform.Data.Doublets.Decorators
2  {
3      using System.Collections.Generic;
4      using Platform.Collections.Arrays;
5      using Platform.Numbers;
6
7      namespace Platform.Data.Doublets.Decorators
8      {
9          public class LinksCascadeUniquenessAndDependenciesResolver<TLink> :
            ↪ LinksUniquenessResolver<TLink>
10          {
11              private static readonly EqualityComparer<TLink> _equalityComparer =
                ↪ EqualityComparer<TLink>.Default;
12
13              public LinksCascadeUniquenessAndDependenciesResolver(ILinks<TLink> links) :
                ↪ base(links) { }
14
15              protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress,
                ↪ TLink newLinkAddress)
16              {
17                  // TODO: Very similar to Merge (logic should be reused)
18                  ulong referencesAsSourceCount = (Integer<TLink>)Links.Count(Constants.Any,
                ↪ oldLinkAddress, Constants.Any);
19                  ulong referencesAsTargetCount = (Integer<TLink>)Links.Count(Constants.Any,
                ↪ Constants.Any, oldLinkAddress);
20                  var references = ArrayPool.Allocate<TLink>((long)(referencesAsSourceCount +
                ↪ referencesAsTargetCount));
21                  var referencesFiller = new ArrayFiller<TLink, TLink>(references,
                ↪ Constants.Continue);
22                  Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any,
                ↪ oldLinkAddress, Constants.Any);
23                  Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any,
                ↪ Constants.Any, oldLinkAddress);
24                  for (ulong i = 0; i < referencesAsSourceCount; i++)
25                  {
26                      var reference = references[i];
27                      if (!_equalityComparer.Equals(reference, oldLinkAddress))
28                      {
29                          Links.Update(reference, newLinkAddress, Links.GetTarget(reference));
30                      }
31                  }
32                  for (var i = (long)referencesAsSourceCount; i < references.Length; i++)
33                  {
34                      var reference = references[i];
35                      if (!_equalityComparer.Equals(reference, oldLinkAddress))
36                      {
37                          Links.Update(reference, Links.GetSource(reference), newLinkAddress);
38                      }
39                  }
40                  ArrayPool.Free(references);
41                  return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
42              }
43          }
44      }
45  }

```

./Decorators/LinksDecoratorBase.cs

```
1  \texttt{
2  using System;
3  using System.Collections.Generic;
4  using Platform.Data.Constants;
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public abstract class LinksDecoratorBase<T> : ILinks<T>
9      {
10         public LinksCombinedConstants<T, T, int> Constants { get; }
11
12         public readonly ILinks<T> Links;
13
14         protected LinksDecoratorBase(ILinks<T> links)
15         {
16             Links = links;
17             Constants = links.Constants;
18         }
19
20         public virtual T Count(IList<T> restriction) => Links.Count(restriction);
21
22         public virtual T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
23             ↪ Links.Each(handler, restrictions);
24
25         public virtual T Create() => Links.Create();
26
27         public virtual T Update(IList<T> restrictions) => Links.Update(restrictions);
28
29         public virtual void Delete(T link) => Links.Delete(link);
30     }
31 }
```

./Decorators/LinksDependenciesValidator.cs

```
1  \texttt{
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets.Decorators
5  {
6      public class LinksDependenciesValidator<T> : LinksDecoratorBase<T>
7      {
8         public LinksDependenciesValidator(ILinks<T> links) : base(links) { }
9
10         public override T Update(IList<T> restrictions)
11         {
12             Links.EnsureNoDependencies(restrictions[IndexPart]);
13             return base.Update(restrictions);
14         }
15
16         public override void Delete(T link)
17         {
18             Links.EnsureNoDependencies(link);
19             base.Delete(link);
20         }
21     }
22 }
23 }
```

./Decorators/LinksDisposableDecoratorBase.cs

```
1  \texttt{
2  using System;
3  using System.Collections.Generic;
```

```
4  using Platform.Disposables;
5  using Platform.Data.Constants;
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      public abstract class LinksDisposableDecoratorBase<T> : DisposableBase, ILinks<T>
10     {
11         public LinksCombinedConstants<T, T, int> Constants { get; }
12
13         public readonly ILinks<T> Links;
14
15         protected LinksDisposableDecoratorBase(ILinks<T> links)
16         {
17             Links = links;
18             Constants = links.Constants;
19         }
20
21         public virtual T Count(IList<T> restriction) => Links.Count(restriction);
22
23         public virtual T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
24             ↪ Links.Each(handler, restrictions);
25
26         public virtual T Create() => Links.Create();
27
28         public virtual T Update(IList<T> restrictions) => Links.Update(restrictions);
29
30         public virtual void Delete(T link) => Links.Delete(link);
31
32         protected override bool AllowMultipleDisposeCalls => true;
33
34         protected override void DisposeCore(bool manual, bool wasDisposed) =>
35             ↪ Disposable.TryDispose(Links);
36     }
37 }
```

./Decorators/LinksInnerReferenceValidator.cs

```
1  \texttt{
2  using System;
3  using System.Collections.Generic;
4
5  namespace Platform.Data.Doublets.Decorators
6  {
7      // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist
8      ↪ or to be external (hybrid link's raw number).
9      public class LinksInnerReferenceValidator<T> : LinksDecoratorBase<T>
10     {
11         public LinksInnerReferenceValidator(ILinks<T> links) : base(links) { }
12
13         public override T Each(Func<IList<T>, T> handler, IList<T> restrictions)
14         {
15             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
16             return base.Each(handler, restrictions);
17         }
18
19         public override T Count(IList<T> restriction)
20         {
21             Links.EnsureInnerReferenceExists(restriction, nameof(restriction));
22             return base.Count(restriction);
23         }
24
25         public override T Update(IList<T> restrictions)
```

```

26 // TODO: Possible values: null, ExistentLink or
    ↳ NonExistentHybrid(ExternalReference)
27 Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
28 return base.Update(restrictions);
29 }
30
31 public override void Delete(T link)
32 {
33 // TODO: Решить считать ли такое исключением, или лишь более конкретным
    ↳ требованием?
34 Links.EnsureLinkExists(link, nameof(link));
35 base.Delete(link);
36 }
37 }
38 }
39 }

```

./Decorators/LinksNonExistentReferencesCreator.cs

```

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

```

./Decorators/LinksNullToSelfReferenceResolver.cs

```

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

```

```

17 public override TLink Update(IList<TLink> restrictions)
18 {
19     restrictions[Constants.SourcePart] =
20     ↳ _equalityComparer.Equals(restrictions[Constants.SourcePart],
21     ↳ Constants.Null) ? restrictions[Constants.IndexPart] :
22     ↳ restrictions[Constants.SourcePart];
23     restrictions[Constants.TargetPart] =
24     ↳ _equalityComparer.Equals(restrictions[Constants.TargetPart],
25     ↳ Constants.Null) ? restrictions[Constants.IndexPart] :
26     ↳ restrictions[Constants.TargetPart];
27     return base.Update(restrictions);
28 }

```

./Decorators/LinksSelfReferenceResolver.cs

```

1  [texttt{
2  using System;
3  using System.Collections.Generic;
4
5  namespace Platform.Data.Doublets.Decorators
6  {
7      public class LinksSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10          ↳ EqualityComparer<TLink>.Default;
11
12         public LinksSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
13
14         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink>
15         ↳ restrictions)
16         {
17             if (!_equalityComparer.Equals(Constants.Any, Constants.Itself)
18                 && (((restrictions.Count > Constants.IndexPart) &&
19                     ↳ _equalityComparer.Equals(restrictions[Constants.IndexPart],
20                     ↳ Constants.Itself))
21                 || ((restrictions.Count > Constants.SourcePart) &&
22                     ↳ _equalityComparer.Equals(restrictions[Constants.SourcePart],
23                     ↳ Constants.Itself))
24                 || ((restrictions.Count > Constants.TargetPart) &&
25                     ↳ _equalityComparer.Equals(restrictions[Constants.TargetPart],
26                     ↳ Constants.Itself))))
27             {
28                 return Constants.Continue;
29             }
30             return base.Each(handler, restrictions);
31         }
32
33         public override TLink Update(IList<TLink> restrictions)
34         {
35             restrictions[Constants.SourcePart] =
36             ↳ _equalityComparer.Equals(restrictions[Constants.SourcePart],
37             ↳ Constants.Itself) ? restrictions[Constants.IndexPart] :
38             ↳ restrictions[Constants.SourcePart];
39             restrictions[Constants.TargetPart] =
40             ↳ _equalityComparer.Equals(restrictions[Constants.TargetPart],
41             ↳ Constants.Itself) ? restrictions[Constants.IndexPart] :
42             ↳ restrictions[Constants.TargetPart];
43         }
44     }
45 }

```

```

29         return base.Update(restrictions);
30     }
31 }
32 }
33 }

```

./Decorators/LinksUniquenessResolver.cs

```

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

```

./Decorators/LinksUniquenessValidator.cs

```

1  \texttt{
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets.Decorators
5  {
6      public class LinksUniquenessValidator<T> : LinksDecoratorBase<T>
7      {
8          public LinksUniquenessValidator(ILinks<T> links) : base(links) { }
9
10         public override T Update(ICollection<T> restrictions)
11         {
12             Links.EnsureDoesNotExists(restrictions[Constants.SourcePart],
13                 ⇨ restrictions[Constants.TargetPart]);
14             return base.Update(restrictions);
15         }
16     }
17 }

```

```

16 }
17 }

```

./Decorators/NonNullContentsLinkDeletionResolver.cs

```

1  \texttt{
2  namespace Platform.Data.Doublets.Decorators
3  {
4      public class NonNullContentsLinkDeletionResolver<T> : LinksDecoratorBase<T>
5      {
6          public NonNullContentsLinkDeletionResolver(ILinks<T> links) : base(links) { }
7
8          public override void Delete(T link)
9          {
10              Links.Update(link, Constants.Null, Constants.Null);
11              base.Delete(link);
12          }
13     }
14 }
15 }

```

./Decorators/UInt64Links.cs

```

1  \texttt{
2  using System;
3  using System.Collections.Generic;
4  using Platform.Collections;
5  using Platform.Collections.Arrays;
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      /// <summary>
10     /// Представляет объект для работы с базой данных (файлом) в формате Links
11     ⇨ (массива взаимосвязей).
12     /// </summary>
13     /// <remarks>
14     /// Возможные оптимизации:
15     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
16     /// + меньше объём БД
17     /// - меньше производительность
18     /// - больше ограничение на количество связей в БД)
19     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования
20     ⇨ БД)
21     /// + меньше объём БД
22     /// - больше сложность
23
24     /// AVL - высота дерева может позволить точно рассчитать размер дерева, нет
25     ⇨ необходимости в SBT.
26     /// AVL дерево можно прошить.
27
28     /// Текущее теоретическое ограничение на размер связей - long.MaxValue
29     /// Желательно реализовать поддержку переключения между деревьями и битовыми
30     ⇨ индексами (битовыми строками) - вариант матрицы (выстраиваемой лениво).
31     ///
32     /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения
33     ⇨ будут выбрасываться только при #if DEBUG
34     /// </remarks>
35     public class UInt64Links : LinksDisposableDecoratorBase<ulong>
36     {
37         public UInt64Links(ILinks<ulong> links) : base(links) { }
38
39         public override ulong Each(Func<ICollection<ulong>, ulong> handler, ICollection<ulong>
40             ⇨ restrictions)
41         {
42             // ...
43         }
44     }
45 }

```

```

35 {
36     this.EnsureLinkIsAnyOrExists(restrictions);
37     return Links.Each(handler, restrictions);
38 }
39
40 public override ulong Create() => Links.CreatePoint();
41
42 public override ulong Update(ICollection<ulong> restrictions)
43 {
44     if (restrictions.IsNullOrEmpty())
45     {
46         return Constants.Null;
47     }
48     // TODO: Remove usages of these hacks (these should not be backwards compatible)
49     if (restrictions.Count == 2)
50     {
51         return this.Merge(restrictions[0], restrictions[1]);
52     }
53     if (restrictions.Count == 4)
54     {
55         return this.UpdateOrCreateOrGet(restrictions[0], restrictions[1], restrictions[2],
56             ↳ restrictions[3]);
57     }
58     // TODO: Looks like this is a common type of exceptions linked with restrictions
59     ↳ support
60     if (restrictions.Count != 3)
61     {
62         throw new NotSupportedException();
63     }
64     var updatedLink = restrictions[Constants.IndexPart];
65     this.EnsureLinkExists(updatedLink, nameof(Constants.IndexPart));
66     var newSource = restrictions[Constants.SourcePart];
67     this.EnsureLinkIsItselfOrExists(newSource, nameof(Constants.SourcePart));
68     var newTarget = restrictions[Constants.TargetPart];
69     this.EnsureLinkIsItselfOrExists(newTarget, nameof(Constants.TargetPart));
70     var existedLink = Constants.Null;
71     if (newSource != Constants.Itself && newTarget != Constants.Itself)
72     {
73         existedLink = this.SearchOrDefault(newSource, newTarget);
74     }
75     if (existedLink == Constants.Null)
76     {
77         var before = Links.GetLink(updatedLink);
78         if (before[Constants.SourcePart] != newSource || before[Constants.TargetPart] !=
79             ↳ newTarget)
80         {
81             Links.Update(updatedLink, newSource == Constants.Itself ? updatedLink :
82                 ↳ newSource,
83                 newTarget == Constants.Itself ? updatedLink : newTarget);
84         }
85         return updatedLink;
86     }
87     else
88     {
89         // Replace one link with another (replaced link is deleted, children are updated
90         ↳ or deleted), it is actually merge operation
91         return this.Merge(updatedLink, existedLink);
92     }
93 }
94
95 /// <summary>Удаляет связь с указанным индексом.</summary>
96 /// <param name="link">Индекс удаляемой связи.</param>

```

```

92 public override void Delete(ulong link)
93 {
94     this.EnsureLinkExists(link);
95     Links.Update(link, Constants.Null, Constants.Null);
96     var referencesCount = Links.Count(Constants.Any, link);
97     if (referencesCount > 0)
98     {
99         var references = new ulong[referencesCount];
100         var referencesFiller = new ArrayFiller<ulong, ulong>(references,
101             ↳ Constants.Continue);
102         Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, link);
103         //references.Sort(); // TODO: Решить необходимо ли для корректного
104         ↳ порядка отмены 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         }
112         //else
113         // TODO: Определить почему здесь есть связи, которых не существует
114         Links.Delete(link);
115     }
116 }
117 }

```

./Decorators/UniLinks.cs

```

1  [texttt{
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using Platform.Collections;
6  using Platform.Collections.Arrays;
7  using Platform.Collections.Lists;
8  using Platform.Helpers.Scopes;
9  using Platform.Data.Constants;
10 using Platform.Data.Universal;
11 using System.Collections.ObjectModel;
12
13 namespace Platform.Data.Doublets.Decorators
14 {
15     /// <remarks>
16     /// What does empty pattern (for condition or substitution) mean? Nothing or
17     ↳ Everything?
18     /// Now we go with nothing. And nothing is something one, but empty, and cannot be
19     ↳ changed by itself. But can cause creation (update from nothing) or deletion (update
20     ↳ to nothing).
21     ///
22     /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
23     ↳ DefaultUniLinksBase, that contains logic itself and can be implemented using both
24     ↳ IDoubletLinks and ILinks.)
25     /// </remarks>
26     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
27     {
28         private static readonly EqualityComparer<TLink> _equalityComparer =
29             ↳ EqualityComparer<TLink>.Default;
30
31         public UniLinks(ILinks<TLink> links) : base(links) { }
32     }
33 }

```

```

27 private struct Transition
28 {
29     public IList<TLink> Before;
30     public IList<TLink> After;
31
32     public Transition(IList<TLink> before, IList<TLink> after)
33     {
34         Before = before;
35         After = after;
36     }
37 }
38
39 public static readonly TLink NullConstant = Use<LinksCombinedConstants<TLink,
↳ TLink, int>>>.Single.Null;
40 public static readonly IReadOnlyList<TLink> NullLink = new
↳ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant,
↳ NullConstant });
41
42 // TODO: Подумать о том, как реализовать древовидный Restriction и
↳ Substitution (Links-Expression)
43 public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>,
↳ TLink> matchedHandler, IList<TLink> substitution, Func<IList<TLink>,
↳ IList<TLink>, TLink> substitutedHandler)
44 {
45     /// List<Transition> transitions = null;
46     if (!restriction.IsNullOrEmpty())
47     {
48         /// Есть причина делать проход (чтение)
49         if (matchedHandler != null)
50         {
51             if (!substitution.IsNullOrEmpty())
52             {
53                 /// restriction => { 0, 0, 0 } | { 0 } // Create
54                 /// substitution => { itself, 0, 0 } | { itself, itself, itself } // Create /
↳ 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
↳ (they are also Null by definition)
60                     var matchDecision = matchedHandler(, NullLink);
61                     if (Equals(matchDecision, Constants.Break))
62                         return false;
63                     if (!Equals(matchDecision, Constants.Skip))
64                         transitions.Add(new Transition(matchedLink, newValue));
65                 }
66             }
67             else
68             {
69                 Func<T, bool> handler;
70                 handler = link =>
71                 {
72                     var matchedLink = Memory.GetLinkValue(link);
73                     var newValue = Memory.GetLinkValue(link);
74                     newValue[Constants.IndexPart] = Constants.Itself;
75                     newValue[Constants.SourcePart] =
↳ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
↳ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
76                     newValue[Constants.TargetPart] =
↳ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
↳ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];

```

```

76     var matchDecision = matchedHandler(matchedLink, newValue);
77     if (Equals(matchDecision, Constants.Break))
78         return false;
79     if (!Equals(matchDecision, Constants.Skip))
80         transitions.Add(new Transition(matchedLink, newValue));
81     return true;
82 };
83 if (!Memory.Each(handler, restriction))
84     return Constants.Break;
85 }
86 }
87 else
88 {
89     Func<T, bool> handler = link =>
90     {
91         var matchedLink = Memory.GetLinkValue(link);
92         var matchDecision = matchedHandler(matchedLink, matchedLink);
93         return !Equals(matchDecision, Constants.Break);
94     };
95     if (!Memory.Each(handler, restriction))
96         return Constants.Break;
97 }
98 }
99 }
100 else
101 {
102     if (substitution != null)
103     {
104         transitions = new List<IList<T>>();
105         Func<T, bool> handler = link =>
106         {
107             var matchedLink = Memory.GetLinkValue(link);
108             transitions.Add(matchedLink);
109             return true;
110         };
111         if (!Memory.Each(handler, restriction))
112             return Constants.Break;
113     }
114     else
115     {
116         return Constants.Continue;
117     }
118 }
119 if (substitution != null)
120 {
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 =>
146 //        ↪ !Equals(matchedHandler(Memory.GetLinkValue(link)), Constants.Break),
147 //        ↪ restriction))
148 //        return Constants.Break;
149 //}
150 //else // Update
151 //{
152 //    // List<IList<T>> matchedLinks = null;
153 //    if (matchedHandler != null)
154 //    {
155 //        matchedLinks = new List<IList<T>>();
156 //        Func<T, bool> handler = link =>
157 //        {
158 //            var matchedLink = Memory.GetLinkValue(link);
159 //            var matchDecision = matchedHandler(matchedLink);
160 //            if (Equals(matchDecision, Constants.Break))
161 //                return false;
162 //            if (!Equals(matchDecision, Constants.Skip))
163 //                matchedLinks.Add(matchedLink);
164 //            return true;
165 //        };
166 //        if (!Memory.Each(handler, restriction))
167 //            return Constants.Break;
168 //    }
169 //    if (!matchedLinks.IsNullOrEmpty())
170 //    {
171 //        var totalMatchedLinks = matchedLinks.Count;
172 //        for (var i = 0; i < totalMatchedLinks; i++)
173 //        {
174 //            var matchedLink = matchedLinks[i];
175 //            if (substitutedHandler != null)
176 //            {
177 //                var newValue = new List<T>(); // TODO: Prepare value to update
178 //                ↪ here
179 //                // TODO: Decide is it actually needed to use Before and After
180 //                ↪ substitution handling.
181 //                var substitutedDecision = substitutedHandler(matchedLink, newValue);
182 //                if (Equals(substitutedDecision, Constants.Break))
183 //                    return Constants.Break;
184 //                if (Equals(substitutedDecision, Constants.Continue))
185 //                {
186 //                    // Actual update here
187 //                    Memory.SetLinkValue(newValue);
188 //                }
189 //                if (Equals(substitutedDecision, Constants.Skip))
190 //                {
191 //                    // Cancel the update. TODO: decide use separate Cancel constant
192 //                    ↪ or Skip is enough?
193 //                }
194 //            }
195 //        }
196 //    }
197 //}
198 //}
199 //}
200 //}
201 //}
202 //}
203 //}
204 //}
205 //}
206 //}
207 //}
208 //}
209 //}
210 //}
211 //}
212 //}
213 //}
214 //}
215 //}
216 //}
217 //}
218 //}
219 //}
220 //}
221 //}
222 //}
223 //}
224 //}
225 //}
226 //}
227 //}
228 //}
229 //}
230 //}
231 //}
232 //}
233 //}
234 //}
235 //}
236 //}
237 //}
238 //}
239 //}
240 //}
241 //}
242 //}
243 //}
244 //}
245 //}
246 //}
247 //}
248 //}
249 //}
250 //}
251 //}
252 //}
253 //}
254 //}
255 //}
256 //}
257 //}
258 //}
259 //}
260 //}
261 //}
262 //}
263 //}
264 //}
265 //}
266 //}
267 //}
268 //}
269 //}
270 //}
271 //}
272 //}
273 //}
274 //}
275 //}
276 //}
277 //}
278 //}
279 //}
280 //}
281 //}
282 //}
283 //}
284 //}
285 //}
286 //}
287 //}
288 //}
289 //}
290 //}
291 //}
292 //}
293 //}
294 //}
295 //}
296 //}
297 //}
298 //}
299 //}
300 //}
301 //}
302 //}
303 //}
304 //}
305 //}
306 //}
307 //}
308 //}
309 //}
310 //}
311 //}
312 //}
313 //}
314 //}
315 //}
316 //}
317 //}
318 //}
319 //}
320 //}
321 //}
322 //}
323 //}
324 //}
325 //}
326 //}
327 //}
328 //}
329 //}
330 //}
331 //}
332 //}
333 //}
334 //}
335 //}
336 //}
337 //}
338 //}
339 //}
340 //}
341 //}
342 //}
343 //}
344 //}
345 //}
346 //}
347 //}
348 //}
349 //}
350 //}
351 //}
352 //}
353 //}
354 //}
355 //}
356 //}
357 //}
358 //}
359 //}
360 //}
361 //}
362 //}
363 //}
364 //}
365 //}
366 //}
367 //}
368 //}
369 //}
370 //}
371 //}
372 //}
373 //}
374 //}
375 //}
376 //}
377 //}
378 //}
379 //}
380 //}
381 //}
382 //}
383 //}
384 //}
385 //}
386 //}
387 //}
388 //}
389 //}
390 //}
391 //}
392 //}
393 //}
394 //}
395 //}
396 //}
397 //}
398 //}
399 //}
400 //}
401 //}
402 //}
403 //}
404 //}
405 //}
406 //}
407 //}
408 //}
409 //}
410 //}
411 //}
412 //}
413 //}
414 //}
415 //}
416 //}
417 //}
418 //}
419 //}
420 //}
421 //}
422 //}
423 //}
424 //}
425 //}
426 //}
427 //}
428 //}
429 //}
430 //}
431 //}
432 //}
433 //}
434 //}
435 //}
436 //}
437 //}
438 //}
439 //}
440 //}
441 //}
442 //}
443 //}
444 //}
445 //}
446 //}
447 //}
448 //}
449 //}
450 //}
451 //}
452 //}
453 //}
454 //}
455 //}
456 //}
457 //}
458 //}
459 //}
460 //}
461 //}
462 //}
463 //}
464 //}
465 //}
466 //}
467 //}
468 //}
469 //}
470 //}
471 //}
472 //}
473 //}
474 //}
475 //}
476 //}
477 //}
478 //}
479 //}
480 //}
481 //}
482 //}
483 //}
484 //}
485 //}
486 //}
487 //}
488 //}
489 //}
490 //}
491 //}
492 //}
493 //}
494 //}
495 //}
496 //}
497 //}
498 //}
499 //}
500 //}
501 //}
502 //}
503 //}
504 //}
505 //}
506 //}
507 //}
508 //}
509 //}
510 //}
511 //}
512 //}
513 //}
514 //}
515 //}
516 //}
517 //}
518 //}
519 //}
520 //}
521 //}
522 //}
523 //}
524 //}
525 //}
526 //}
527 //}
528 //}
529 //}
530 //}
531 //}
532 //}
533 //}
534 //}
535 //}
536 //}
537 //}
538 //}
539 //}
540 //}
541 //}
542 //}
543 //}
544 //}
545 //}
546 //}
547 //}
548 //}
549 //}
550 //}
551 //}
552 //}
553 //}
554 //}
555 //}
556 //}
557 //}
558 //}
559 //}
560 //}
561 //}
562 //}
563 //}
564 //}
565 //}
566 //}
567 //}
568 //}
569 //}
570 //}
571 //}
572 //}
573 //}
574 //}
575 //}
576 //}
577 //}
578 //}
579 //}
580 //}
581 //}
582 //}
583 //}
584 //}
585 //}
586 //}
587 //}
588 //}
589 //}
590 //}
591 //}
592 //}
593 //}
594 //}
595 //}
596 //}
597 //}
598 //}
599 //}
600 //}
601 //}
602 //}
603 //}
604 //}
605 //}
606 //}
607 //}
608 //}
609 //}
610 //}
611 //}
612 //}
613 //}
614 //}
615 //}
616 //}
617 //}
618 //}
619 //}
620 //}
621 //}
622 //}
623 //}
624 //}
625 //}
626 //}
627 //}
628 //}
629 //}
630 //}
631 //}
632 //}
633 //}
634 //}
635 //}
636 //}
637 //}
638 //}
639 //}
640 //}
641 //}
642 //}
643 //}
644 //}
645 //}
646 //}
647 //}
648 //}
649 //}
650 //}
651 //}
652 //}
653 //}
654 //}
655 //}
656 //}
657 //}
658 //}
659 //}
660 //}
661 //}
662 //}
663 //}
664 //}
665 //}
666 //}
667 //}
668 //}
669 //}
670 //}
671 //}
672 //}
673 //}
674 //}
675 //}
676 //}
677 //}
678 //}
679 //}
680 //}
681 //}
682 //}
683 //}
684 //}
685 //}
686 //}
687 //}
688 //}
689 //}
690 //}
691 //}
692 //}
693 //}
694 //}
695 //}
696 //}
697 //}
698 //}
699 //}
700 //}
701 //}
702 //}
703 //}
704 //}
705 //}
706 //}
707 //}
708 //}
709 //}
710 //}
711 //}
712 //}
713 //}
714 //}
715 //}
716 //}
717 //}
718 //}
719 //}
720 //}
721 //}
722 //}
723 //}
724 //}
725 //}
726 //}
727 //}
728 //}
729 //}
730 //}
731 //}
732 //}
733 //}
734 //}
735 //}
736 //}
737 //}
738 //}
739 //}
740 //}
741 //}
742 //}
743 //}
744 //}
745 //}
746 //}
747 //}
748 //}
749 //}
750 //}
751 //}
752 //}
753 //}
754 //}
755 //}
756 //}
757 //}
758 //}
759 //}
760 //}
761 //}
762 //}
763 //}
764 //}
765 //}
766 //}
767 //}
768 //}
769 //}
770 //}
771 //}
772 //}
773 //}
774 //}
775 //}
776 //}
777 //}
778 //}
779 //}
780 //}
781 //}
782 //}
783 //}
784 //}
785 //}
786 //}
787 //}
788 //}
789 //}
790 //}
791 //}
792 //}
793 //}
794 //}
795 //}
796 //}
797 //}
798 //}
799 //}
800 //}
801 //}
802 //}
803 //}
804 //}
805 //}
806 //}
807 //}
808 //}
809 //}
810 //}
811 //}
812 //}
813 //}
814 //}
815 //}
816 //}
817 //}
818 //}
819 //}
820 //}
821 //}
822 //}
823 //}
824 //}
825 //}
826 //}
827 //}
828 //}
829 //}
830 //}
831 //}
832 //}
833 //}
834 //}
835 //}
836 //}
837 //}
838 //}
839 //}
840 //}
841 //}
842 //}
843 //}
844 //}
845 //}
846 //}
847 //}
848 //}
849 //}
850 //}
851 //}
852 //}
853 //}
854 //}
855 //}
856 //}
857 //}
858 //}
859 //}
860 //}
861 //}
862 //}
863 //}
864 //}
865 //}
866 //}
867 //}
868 //}
869 //}
870 //}
871 //}
872 //}
873 //}
874 //}
875 //}
876 //}
877 //}
878 //}
879 //}
880 //}
881 //}
882 //}
883 //}
884 //}
885 //}
886 //}
887 //}
888 //}
889 //}
890 //}
891 //}
892 //}
893 //}
894 //}
895 //}
896 //}
897 //}
898 //}
899 //}
900 //}
901 //}
902 //}
903 //}
904 //}
905 //}
906 //}
907 //}
908 //}
909 //}
910 //}
911 //}
912 //}
913 //}
914 //}
915 //}
916 //}
917 //}
918 //}
919 //}
920 //}
921 //}
922 //}
923 //}
924 //}
925 //}
926 //}
927 //}
928 //}
929 //}
930 //}
931 //}
932 //}
933 //}
934 //}
935 //}
936 //}
937 //}
938 //}
939 //}
940 //}
941 //}
942 //}
943 //}
944 //}
945 //}
946 //}
947 //}
948 //}
949 //}
950 //}
951 //}
952 //}
953 //}
954 //}
955 //}
956 //}
957 //}
958 //}
959 //}
960 //}
961 //}
962 //}
963 //}
964 //}
965 //}
966 //}
967 //}
968 //}
969 //}
970 //}
971 //}
972 //}
973 //}
974 //}
975 //}
976 //}
977 //}
978 //}
979 //}
980 //}
981 //}
982 //}
983 //}
984 //}
985 //}
986 //}
987 //}
988 //}
989 //}
990 //}
991 //}
992 //}
993 //}
994 //}
995 //}
996 //}
997 //}
998 //}
999 //}
1000 //}

```

```

}
}

public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>,
    ↪ TLink> matchHandler, IList<TLink> substitution, Func<IList<TLink>,
    ↪ IList<TLink>, TLink> substitutionHandler)
{
    if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
    {
        return Constants.Continue;
    }
    else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
    {
        ↪ Check if it is a correct condition
        // Or it only applies to trigger without matchHandler.
        throw new NotImplementedException();
    }
    else if (!substitution.IsNullOrEmpty()) // Creation
    {
        var before = ArrayPool<TLink>.Empty;
        // Что должно означать False здесь? Остановиться (перестать идти) или
        ↪ пропустить (пройти мимо) или пустить (взять)?
        if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
            ↪ Constants.Break))
        {
            return Constants.Break;
        }
        var after = (IList<TLink>)substitution.ToArray();
        if (_equalityComparer.Equals(after[0], default))
        {
            var newLink = Links.Create();
            after[0] = newLink;
        }
        if (substitution.Count == 1)
        {
            after = Links.GetLink(substitution[0]);
        }
        else if (substitution.Count == 3)
        {
            Links.Update(after);
        }
        else
        {
            throw new NotSupportedException();
        }
        if (matchHandler != null)
        {
            return substitutionHandler(before, after);
        }
        return Constants.Continue;
    }
    else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
    {
        if (patternOrCondition.Count == 1)
        {
            var linkToDelete = patternOrCondition[0];
            var before = Links.GetLink(linkToDelete);
            if (matchHandler != null &&
                ↪ _equalityComparer.Equals(matchHandler(before), Constants.Break))
            {
                return Constants.Break;
            }
        }
    }
}

```

```

249         var after = ArrayPool<TLink>.Empty;
250         Links.Update(linkToDelete, Constants.Null, Constants.Null);
251         Links.Delete(linkToDelete);
252         if (matchHandler != null)
253         {
254             return substitutionHandler(before, after);
255         }
256         return Constants.Continue;
257     }
258     else
259     {
260         throw new NotSupportedException();
261     }
262 }
263 else // Replace / Update
264 {
265     if (patternOrCondition.Count == 1) //-V3125
266     {
267         var linkToUpdate = patternOrCondition[0];
268         var before = Links.GetLink(linkToUpdate);
269         if (matchHandler != null &&
270             ↪ _equalityComparer.Equals(matchHandler(before), Constants.Break))
271         {
272             return Constants.Break;
273         }
274         var after = (IList<TLink>)substitution.ToArray(); //-V3125
275         if (_equalityComparer.Equals(after[0], default))
276         {
277             after[0] = linkToUpdate;
278         }
279         if (substitution.Count == 1)
280         {
281             if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
282             {
283                 after = Links.GetLink(substitution[0]);
284                 Links.Update(linkToUpdate, Constants.Null, Constants.Null);
285                 Links.Delete(linkToUpdate);
286             }
287         }
288         else if (substitution.Count == 3)
289         {
290             Links.Update(after);
291         }
292         else
293         {
294             throw new NotSupportedException();
295         }
296         if (matchHandler != null)
297         {
298             return substitutionHandler(before, after);
299         }
300         return Constants.Continue;
301     }
302     else
303     {
304         throw new NotSupportedException();
305     }
306 }
307
308 /// <remarks>
309 /// IList[IList[IList[T]]]

```

```

310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333

```

```

public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
    ↪ substitution)
{
    var changes = new List<IList<IList<TLink>>>();
    Trigger(condition, AlwaysContinue, substitution, (before, after) =>
    {
        var change = new[] { before, after };
        changes.Add(change);
        return Constants.Continue;
    });
    return changes;
}

private TLink AlwaysContinue(IList<TLink> linkToMatch) => Constants.Continue;
}
}
}

```

./DoubletComparer.cs

```

1  \texttt{
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Data.Doublets
6  {
7      /// <remarks>
8      /// TODO: Может стоит попробовать ref во всех методах (IRefEqualityComparer)
9      /// 2x faster with comparer
10     /// </remarks>
11     public class DoubletComparer<T> : IEqualityComparer<Doublet<T>>
12     {
13         private static readonly EqualityComparer<T> _equalityComparer =
14             ↪ EqualityComparer<T>.Default;
15
16         public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public bool Equals(Doublet<T> x, Doublet<T> y) =>
20             ↪ _equalityComparer.Equals(x.Source, y.Source) &&
21             ↪ _equalityComparer.Equals(x.Target, y.Target);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public int GetHashCode(Doublet<T> obj) => unchecked(obj.Source.GetHashCode()
25             ↪ << 15 ^ obj.Target.GetHashCode());
26     }
27 }
28
29
30
31
32
33
34
35
36
37
38
39

```

./Doublet.cs

```

1  \texttt{
2  using System;
3  using System.Collections.Generic;
4

```

```

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

```

./Hybrid.cs

```

1  [texttt{
2  using System;
3  using System.Reflection;
4  using Platform.Reflection;
5  using Platform.Converters;
6  using Platform.Numbers;
7
8  namespace Platform.Data.Doublets
9  {
10     public class Hybrid<T>
11     {
12         public readonly T Value;
13         public bool IsNothing => Convert.ToInt64(To.Signed(Value)) == 0;
14         public bool IsInternal => Convert.ToInt64(To.Signed(Value)) > 0;
15         public bool IsExternal => Convert.ToInt64(To.Signed(Value)) < 0;
16         public long AbsoluteValue => Math.Abs(Convert.ToInt64(To.Signed(Value)));
17
18         public Hybrid(T value)
19         {
20             if (CachedTypeInfo<T>.IsSigned)
21             {
22                 throw new NotSupportedException();
23             }
24             Value = value;
25         }
26
27         public Hybrid(object value) => Value =
28             ↳ To.UnsignedAs<T>(Convert.ChangeType(value,
29             ↳ CachedTypeInfo<T>.SignedVersion));
30
31         public Hybrid(object value, bool isExternal)
32         {
33             var signedType = CachedTypeInfo<T>.SignedVersion;
34             var signedValue = Convert.ChangeType(value, signedType);
35             var abs = typeof(MathHelpers).GetTypeInfo().GetMethod("Abs").MakeGenericM
36                 ↳ ethod(signedType);
37             var negate = typeof(MathHelpers).GetTypeInfo().GetMethod("Negate").MakeGen
38                 ↳ ethod(signedType);

```

```

35     var absoluteValue = abs.Invoke(null, new[] { signedValue });
36     var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
37         ↳ 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);
48
49 public static explicit operator Hybrid<T>(int integer) => new Hybrid<T>(integer);
50
51 public static explicit operator Hybrid<T>(ushort integer) => new
52     ↳ Hybrid<T>(integer);
53
54 public static explicit operator Hybrid<T>(short integer) => new Hybrid<T>(integer);
55
56 public static explicit operator Hybrid<T>(byte integer) => new Hybrid<T>(integer);
57
58 public static explicit operator Hybrid<T>(sbyte integer) => new Hybrid<T>(integer);
59
60 public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;
61
62 public static explicit operator ulong(Hybrid<T> hybrid) =>
63     ↳ Convert.ToUInt64(hybrid.Value);
64
65 public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;
66
67 public static explicit operator uint(Hybrid<T> hybrid) =>
68     ↳ Convert.ToUInt32(hybrid.Value);
69
70 public static explicit operator int(Hybrid<T> hybrid) =>
71     ↳ Convert.ToInt32(hybrid.AbsoluteValue);
72
73 public static explicit operator ushort(Hybrid<T> hybrid) =>
74     ↳ Convert.ToUInt16(hybrid.Value);
75
76 public static explicit operator short(Hybrid<T> hybrid) =>
77     ↳ Convert.ToInt16(hybrid.AbsoluteValue);
78
79 public static explicit operator byte(Hybrid<T> hybrid) =>
80     ↳ Convert.ToByte(hybrid.Value);
81
82 public static explicit operator sbyte(Hybrid<T> hybrid) =>
83     ↳ Convert.ToSByte(hybrid.AbsoluteValue);
84
85 public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
86     ↳ default(T).ToString() : IsExternal ? $"{<{AbsoluteValue}>" : Value.ToString();
87 }
88 }
89 }

```

./ILinks.cs

```

1  [texttt{
2  using Platform.Data.Constants;
3
4  namespace Platform.Data.Doublets
5  {

```

```

6     public interface ILinks<TLink> : ILinks<TLink, LinksCombinedConstants<TLink,
7         ↳ TLink, int>>
8     {
9     }
10 }

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

```

```

46         Integer<TLink> link = RandomHelpers.Default.NextUInt64(linksAddressRange);
47         links.Delete(link);
48         if ((Integer<TLink>)links.Count() < min)
49         {
50             break;
51         }
52     }
53 }
54
55 /// <remarks>
56 /// TODO: Возможно есть очень простой способ это сделать.
57 /// (Например просто удалить файл, или изменить его размер таким образом,
58 /// чтобы удалился весь контент)
59 /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
60 /// </remarks>
61 public static void DeleteAll<TLink>(this ILinks<TLink> links)
62 {
63     var equalityComparer = EqualityComparer<TLink>.Default;
64     var comparer = Comparer<TLink>.Default;
65     for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
66         ↳ ArithmeticHelpers.Decrement(i))
67     {
68         links.Delete(i);
69         if (!equalityComparer.Equals(links.Count(), ArithmeticHelpers.Decrement(i)))
70         {
71             i = links.Count();
72         }
73     }
74
75     public static TLink First<TLink>(this ILinks<TLink> links)
76     {
77         TLink firstLink = default;
78         var equalityComparer = EqualityComparer<TLink>.Default;
79         if (equalityComparer.Equals(links.Count(), default))
80         {
81             throw new Exception("В хранилище нет связей.");
82         }
83         links.Each(links.Constants.Any, links.Constants.Any, link =>
84         {
85             firstLink = link[links.Constants.IndexPart];
86             return links.Constants.Break;
87         });
88         if (equalityComparer.Equals(firstLink, default))
89         {
90             throw new Exception("В процессе поиска по хранилищу не было найдено
91                 ↳ связей.");
92         }
93         return firstLink;
94     }
95
96     public static bool IsInnerReference<TLink>(this ILinks<TLink> links, TLink
97         ↳ reference)
98     {
99         var constants = links.Constants;
100        var comparer = Comparer<TLink>.Default;
101        return comparer.Compare(constants.MinPossibleIndex, reference) >= 0 &&
102            ↳ comparer.Compare(reference, constants.MaxPossibleIndex) <= 0;
103    }

```

```

#region Paths

```

```

104 /// <remarks>
105 /// TODO: Как так? Как то что ниже может быть корректно?
106 /// Скорее всего практически не применимо
107 /// Предполагалось, что можно было конвертировать формируемый в проходе
    ↪ через SequenceWalker
108 /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
109 /// TODO: Возможно нужен метод, который именно выбрасывает исключения
    ↪ (EnsurePathExists)
110 /// </remarks>
111 public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params
    ↪ TLink[] path)
112 {
113     var current = path[0];
114     //EnsureLinkExists(current, "path");
115     if (!links.Exists(current))
116     {
117         return false;
118     }
119     var equalityComparer = EqualityComparer<TLink>.Default;
120     var constants = links.Constants;
121     for (var i = 1; i < path.Length; i++)
122     {
123         var next = path[i];
124         var values = links.GetLink(current);
125         var source = values[constants.SourcePart];
126         var target = values[constants.TargetPart];
127         if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
            ↪ next))
128         {
129             //throw new Exception(string.Format("Невозможно выбрать путь, так как
            ↪ и Source и Target совпадают с элементом пути {0}.", next));
            return false;
130         }
131         if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
            ↪ target))
132         {
133             //throw new Exception(string.Format("Невозможно продолжить путь через
            ↪ элемент пути {0}", next));
            return false;
134         }
135         current = next;
136     }
137     return true;
138 }
139
140 /// <remarks>
141 /// Может потребовать дополнительного стека для PathElement's при
    ↪ использовании SequenceWalker.
142 /// </remarks>
143 public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root,
    ↪ params int[] path)
144 {
145     links.EnsureLinkExists(root, "root");
146     var currentLink = root;
147     for (var i = 0; i < path.Length; i++)
148     {
149         currentLink = links.GetLink(currentLink)[path[i]];
150     }
151     return currentLink;
152 }
153
154 }
155

```

```

156 public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this
    ↪ ILinks<TLink> links, TLink root, ulong size, ulong index)
157 {
158     var constants = links.Constants;
159     var source = constants.SourcePart;
160     var target = constants.TargetPart;
161     if (!MathHelpers.IsPowerOfTwo(size))
162     {
163         throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes
            ↪ other than powers of two are not supported.");
164     }
165     var path = new BitArray(BitConverter.GetBytes(index));
166     var length = BitwiseHelpers.GetLowestBitPosition(size);
167     links.EnsureLinkExists(root, "root");
168     var currentLink = root;
169     for (var i = length - 1; i >= 0; i--)
170     {
171         currentLink = links.GetLink(currentLink)[path[i] ? target : source];
172     }
173     return currentLink;
174 }
175
176 #endregion
177
178 /// <summary>
179 /// Возвращает индекс указанной связи.
180 /// </summary>
181 /// <param name="links">Хранилище связей.</param>
182 /// <param name="link">Связь представленная списком, состоящим из её адреса
    ↪ и содержимого.</param>
183 /// <returns>Индекс начальной связи для указанной связи.</returns>
184 [MethodImpl(MethodImplOptions.AggressiveInlining)]
185 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link)
    ↪ => link[links.Constants.IndexPart];
186
187 /// <summary>
188 /// Возвращает индекс начальной (Source) связи для указанной связи.
189 /// </summary>
190 /// <param name="links">Хранилище связей.</param>
191 /// <param name="link">Индекс связи.</param>
192 /// <returns>Индекс начальной связи для указанной связи.</returns>
193 [MethodImpl(MethodImplOptions.AggressiveInlining)]
194 public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
    ↪ links.GetLink(link)[links.Constants.SourcePart];
195
196 /// <summary>
197 /// Возвращает индекс начальной (Source) связи для указанной связи.
198 /// </summary>
199 /// <param name="links">Хранилище связей.</param>
200 /// <param name="link">Связь представленная списком, состоящим из её адреса
    ↪ и содержимого.</param>
201 /// <returns>Индекс начальной связи для указанной связи.</returns>
202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink>
    ↪ link) => link[links.Constants.SourcePart];
204
205 /// <summary>
206 /// Возвращает индекс конечной (Target) связи для указанной связи.
207 /// </summary>
208 /// <param name="links">Хранилище связей.</param>
209 /// <param name="link">Индекс связи.</param>
210 /// <returns>Индекс конечной связи для указанной связи.</returns>

```

```

211 [MethodImpl(MethodImplOptions.AggressiveInlining)]
212 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
    ↳ links.GetLink(link)[links.Constants.TargetPart];

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
218 ↳ и содержимого.</param>
219 /// <returns>Индекс конечной связи для указанной связи.</returns>
220 [MethodImpl(MethodImplOptions.AggressiveInlining)]
221 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink>
    ↳ link) => link[links.Constants.TargetPart];

222 /// <summary>
223 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая
224 ↳ обработчик (handler) для каждой подходящей связи.
225 /// </summary>
226 /// <param name="links">Хранилище связей.</param>
227 /// <param name="handler">Обработчик каждой подходящей связи.</param>
228 /// <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>
229 /// <returns>True, в случае если проход по связям не был прерван и False в
    ↳ обратном случае.</returns>
230 [MethodImpl(MethodImplOptions.AggressiveInlining)]
231 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>,
    ↳ TLink> handler, params TLink[] restrictions)
    ↳ => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
    ↳ links.Constants.Continue);

232 /// <summary>
233 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая
234 ↳ 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     var constants = links.Constants;
244     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
    ↳ constants.Break, constants.Any, source, target);
245 }
246
247 /// <summary>
248 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая
249 ↳ обработчик (handler) для каждой подходящей связи.
250 /// </summary>
251

```

```

252 /// <param name="links">Хранилище связей.</param>
253 /// <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>
254 /// <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>
255 /// <param name="handler">Обработчик каждой подходящей связи.</param>
256 /// <returns>True, в случае если проход по связям не был прерван и False в
    ↳ обратном случае.</returns>
257 [MethodImpl(MethodImplOptions.AggressiveInlining)]
258 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target, Func<IList<TLink>, TLink> handler)
    ↳ {
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     var constants = links.Constants;
266     int listSize = (Integer<TLink>)links.Count(restrictions);
267     var list = new IList<TLink>[listSize];
268     if (listSize > 0)
269     {
270         var filler = new ArrayFiller<IList<TLink>, TLink>(list,
    ↳ ↳ links.Constants.Continue);
271         links.Each(filler.AddAndReturnConstant, restrictions);
272     }
273     return list;
274 }
275
276 /// <summary>
277 /// Возвращает значение, определяющее существует ли связь с указанными
278 ↳ началом и концом в хранилище связей.
279 /// </summary>
280 /// <param name="links">Хранилище связей.</param>
281 /// <param name="source">Начало связи.</param>
282 /// <param name="target">Конец связи.</param>
283 /// <returns>Значение, определяющее существует ли связь.</returns>
284 [MethodImpl(MethodImplOptions.AggressiveInlining)]
285 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target) =>
    ↳ Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source,
    ↳ target), default) > 0;
286
287 #region Ensure
288 // TODO: May be move to EnsureExtensions or make it both there and here
289
290 [MethodImpl(MethodImplOptions.AggressiveInlining)]
291 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
    ↳ TLink reference, string argumentName)
    ↳ {
292     if (links.IsInnerReference(reference) && !links.Exists(reference))
293     {
294         throw new ArgumentLinkDoesNotExistsException<TLink>(reference,
    ↳ ↳ argumentName);
295     }
296 }
297
298

```

```

299 [MethodImpl(MethodImplOptions.AggressiveInlining)]
300 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
301 ↪ IList<TLink> restrictions, string argumentName)
302 {
303     for (int i = 0; i < restrictions.Count; i++)
304     {
305         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
306     }
307 }
308
309 [MethodImpl(MethodImplOptions.AggressiveInlining)]
310 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links,
311 ↪ IList<TLink> restrictions)
312 {
313     for (int i = 0; i < restrictions.Count; i++)
314     {
315         links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
316     }
317 }
318
319 [MethodImpl(MethodImplOptions.AggressiveInlining)]
320 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links,
321 ↪ TLink link, string argumentName)
322 {
323     var equalityComparer = EqualityComparer<TLink>.Default;
324     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
325     {
326         throw new ArgumentLinkDoesNotExistsException<TLink>(link,
327 ↪ argumentName);
328     }
329 }
330
331 [MethodImpl(MethodImplOptions.AggressiveInlining)]
332 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links,
333 ↪ TLink link, string argumentName)
334 {
335     var equalityComparer = EqualityComparer<TLink>.Default;
336     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
337     {
338         throw new ArgumentLinkDoesNotExistsException<TLink>(link,
339 ↪ argumentName);
340     }
341 }
342
343 [MethodImpl(MethodImplOptions.AggressiveInlining)]
344 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink
345 ↪ source, TLink target)
346 {
347     if (links.Exists(source, target))
348     {
349         throw new LinkWithSameValueAlreadyExistsException();
350     }
351 }
352
353 [MethodImpl(MethodImplOptions.AggressiveInlining)]
354 public static void EnsureNoDependencies<TLink>(this ILinks<TLink> links, TLink
355 ↪ link)
356 {
357     if (links.DependenciesExist(link))
358     {
359

```

```

353         throw new ArgumentLinkHasDependenciesException<TLink>(link);
354     }
355 }
356
357 [MethodImpl(MethodImplOptions.AggressiveInlining)]
358 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
359 ↪ addresses) => links.EnsureCreated(links.Create, addresses);
360
361 [MethodImpl(MethodImplOptions.AggressiveInlining)]
362 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params
363 ↪ TLink[] addresses) => links.EnsureCreated(links.CreatePoint, addresses);
364
365 [MethodImpl(MethodImplOptions.AggressiveInlining)]
366 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink>
367 ↪ creator, params TLink[] addresses)
368 {
369     var constants = links.Constants;
370     var nonExistentAddresses = new HashSet<ulong>(addresses.Where(x =>
371 ↪ !links.Exists(x)).Select(x => (ulong)(Integer<TLink>)x));
372     if (nonExistentAddresses.Count > 0)
373     {
374         var max = nonExistentAddresses.Max();
375         // TODO: Эту верхнюю границу нужно разрешить переопределять
376         ↪ (проверить применяется ли эта логика)
377         max = Math.Min(max, (Integer<TLink>)constants.MaxPossibleIndex);
378         var createdLinks = new List<TLink>();
379         var equalityComparer = EqualityComparer<TLink>.Default;
380         TLink createdLink = creator();
381         while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
382         {
383             createdLinks.Add(createdLink);
384         }
385         for (var i = 0; i < createdLinks.Count; i++)
386         {
387             if (!nonExistentAddresses.Contains((Integer<TLink>)createdLinks[i]))
388             {
389                 links.Delete(createdLinks[i]);
390             }
391         }
392     }
393 }
394
395 #endregion
396
397 [MethodImpl(MethodImplOptions.AggressiveInlining)]
398 public static ulong DependenciesCount<TLink>(this ILinks<TLink> links, TLink link)
399 {
400     var constants = links.Constants;
401     var values = links.GetLink(link);
402     ulong referencesAsSource = (Integer<TLink>)links.Count(constants.Any, link,
403 ↪ constants.Any);
404     var equalityComparer = EqualityComparer<TLink>.Default;
405     if (equalityComparer.Equals(values[constants.SourcePart], link))
406     {
407         referencesAsSource--;
408     }
409     ulong referencesAsTarget = (Integer<TLink>)links.Count(constants.Any,
410 ↪ constants.Any, link);
411     if (equalityComparer.Equals(values[constants.TargetPart], link))
412     {
413         referencesAsTarget--;
414     }
415 }

```

```

407     }
408     return referencesAsSource + referencesAsTarget;
409 }
410
411 /// <param name="links">Хранилище связей.</param>
412 [MethodImpl(MethodImplOptions.AggressiveInlining)]
413 public static bool DependenciesExist<TLink>(this ILinks<TLink> links, TLink link)
414     ↳ => links.DependenciesCount(link) > 0;
415
416 /// <param name="links">Хранилище связей.</param>
417 [MethodImpl(MethodImplOptions.AggressiveInlining)]
418 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink
419     ↳ source, TLink target)
420 {
421     var constants = links.Constants;
422     var values = links.GetLink(link);
423     var equalityComparer = EqualityComparer<TLink>.Default;
424     return equalityComparer.Equals(values[constants.SourcePart], source) &&
425         ↳ equalityComparer.Equals(values[constants.TargetPart], target);
426 }
427
428 /// <summary>
429 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
430 /// </summary>
431 /// <param name="links">Хранилище связей.</param>
432 /// <param name="source">Индекс связи, которая является началом для искомой
433     ↳ связи.</param>
434 /// <param name="target">Индекс связи, которая является концом для искомой
435     ↳ связи.</param>
436 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
437     ↳ (концом).</returns>
438 [MethodImpl(MethodImplOptions.AggressiveInlining)]
439 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink
440     ↳ source, TLink target)
441 {
442     var constants = links.Constants;
443     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break,
444         ↳ default);
445     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
446     return setter.Result;
447 }
448
449 /// <param name="links">Хранилище связей.</param>
450 [MethodImpl(MethodImplOptions.AggressiveInlining)]
451 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
452 {
453     var link = links.Create();
454     return links.Update(link, link, link);
455 }
456
457 /// <param name="links">Хранилище связей.</param>
458 [MethodImpl(MethodImplOptions.AggressiveInlining)]
459 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink
460     ↳ source, TLink target) => links.Update(links.Create(), source, target);
461
462 /// <summary>
463 /// Обновляет связь с указанными началом (Source) и концом (Target)
464     ↳ на связь с указанными началом (NewSource) и концом (NewTarget).
465 /// </summary>
466 /// <param name="links">Хранилище связей.</param>
467 /// <param name="link">Индекс обновляемой связи.</param>

```

```

459     /// <param name="newSource">Индекс связи, которая является началом связи,
460     ↳ на которую выполняется обновление.</param>
461     /// <param name="newTarget">Индекс связи, которая является концом связи, на
462     ↳ которую выполняется 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
466     ↳ newSource, TLink newTarget) => links.Update(new[] { link, newSource,
467     ↳ newTarget });
468
469 /// <summary>
470 /// Обновляет связь с указанными началом (Source) и концом (Target)
471     ↳ на связь с указанными началом (NewSource) и концом (NewTarget).
472 /// </summary>
473 /// <param name="links">Хранилище связей.</param>
474 /// <param name="restrictions">Ограничения на содержимое связей. Каждое
475     ↳ ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая
476     ↳ ссылку на пустоту, Itself - требование установить ссылку на себя, 1..∞
477     ↳ конкретный адрес другой связи.</param>
478 /// <returns>Индекс обновлённой связи.</returns>
479 [MethodImpl(MethodImplOptions.AggressiveInlining)]
480 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[]
481     ↳ restrictions)
482 {
483     if (restrictions.Length == 2)
484     {
485         return links.Merge(restrictions[0], restrictions[1]);
486     }
487     if (restrictions.Length == 4)
488     {
489         return links.UpdateOrCreateGet(restrictions[0], restrictions[1], restrictions[2],
490             ↳ restrictions[3]);
491     }
492     else
493     {
494         return links.Update(restrictions);
495     }
496 }
497
498 /// <summary>
499 /// Создаёт связь (если она не существовала), либо возвращает индекс
500     ↳ существующей связи с указанными Source (началом) и Target (концом).
501 /// </summary>
502 /// <param name="links">Хранилище связей.</param>
503 /// <param name="source">Индекс связи, которая является началом на
504     ↳ создаваемой связи.</param>
505 /// <param name="target">Индекс связи, которая является концом для
506     ↳ создаваемой связи.</param>
507 /// <returns>Индекс связи, с указанным Source (началом) и Target
508     ↳ (концом)</returns>
509 [MethodImpl(MethodImplOptions.AggressiveInlining)]
510 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source,
511     ↳ TLink target)
512 {
513     var link = links.SearchOrDefault(source, target);
514     if (EqualityComparer<TLink>.Default.Equals(link, default))
515     {
516         link = links.CreateAndUpdate(source, target);
517     }
518     return link;
519 }

```



```

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
512   ↳ обновляемой связи.</param>
513 /// <param name="target">Индекс связи, которая является концом обновляемой
514   ↳ связи.</param>
515 /// <param name="newSource">Индекс связи, которая является началом связи,
516   ↳ на которую выполняется обновление.</param>
517 /// <param name="newTarget">Индекс связи, которая является концом связи, на
518   ↳ которую выполняется обновление.</param>
519 /// <returns>Индекс обновлённой связи.</returns>
520 [MethodImpl(MethodImplOptions.AggressiveInlining)]
521 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink
522   ↳ source, TLink target, TLink newSource, TLink newTarget)
523 {
524     var equalityComparer = EqualityComparer<TLink>.Default;
525     var link = links.SearchOrDefault(source, target);
526     if (equalityComparer.Equals(link, default))
527     {
528         return links.CreateAndUpdate(newSource, newTarget);
529     }
530     if (equalityComparer.Equals(newSource, source) &&
531         ↳ equalityComparer.Equals(newTarget, target))
532     {
533         return link;
534     }
535     return links.Update(link, newSource, newTarget);
536 }
537 /// <summary>Удаляет связь с указанными началом (Source) и концом
538   ↳ (Target).</summary>
539 /// <param name="links">Хранилище связей.</param>
540 /// <param name="source">Индекс связи, которая является началом удаляемой
541   ↳ связи.</param>
542 /// <param name="target">Индекс связи, которая является концом удаляемой
543   ↳ связи.</param>
544 [MethodImpl(MethodImplOptions.AggressiveInlining)]
545 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source,
546   ↳ TLink target)
547 {
548     var link = links.SearchOrDefault(source, target);
549     if (!EqualityComparer<TLink>.Default.Equals(link, default))
550     {
551         links.Delete(link);
552         return link;
553     }
554     return default;
555 }
556 /// <summary>Удаляет несколько связей.</summary>
557 /// <param name="links">Хранилище связей.</param>
558 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
559 [MethodImpl(MethodImplOptions.AggressiveInlining)]
560 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink>
561   ↳ deletedLinks)
562 {
563     for (int i = 0; i < deletedLinks.Count; i++)

```

```

556 {
557     links.Delete(deletedLinks[i]);
558 }
559 }
560 // Replace one link with another (replaced link is deleted, children are updated or
561   ↳ deleted)
562 public static TLink Merge<TLink>(this ILinks<TLink> links, TLink linkIndex,
563   ↳ TLink newLink)
564 {
565     var equalityComparer = EqualityComparer<TLink>.Default;
566     if (equalityComparer.Equals(linkIndex, newLink))
567     {
568         return newLink;
569     }
570     var constants = links.Constants;
571     ulong referencesAsSourceCount = (Integer<TLink>)links.Count(constants.Any,
572       ↳ linkIndex, constants.Any);
573     ulong referencesAsTargetCount = (Integer<TLink>)links.Count(constants.Any,
574       ↳ constants.Any, linkIndex);
575     var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(linkIndex)) &&
576       ↳ referencesAsSourceCount == 1 && referencesAsTargetCount == 1;
577     if (!isStandalonePoint)
578     {
579         var totalReferences = referencesAsSourceCount + referencesAsTargetCount;
580         if (totalReferences > 0)
581         {
582             var references = ArrayPool.Allocate<TLink>((long)totalReferences);
583             var referencesFiller = new ArrayFiller<TLink, TLink>(references,
584               ↳ links.Constants.Continue);
585             links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any,
586               ↳ linkIndex, constants.Any);
587             links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any,
588               ↳ constants.Any, linkIndex);
589             for (ulong i = 0; i < referencesAsSourceCount; i++)
590             {
591                 var reference = references[i];
592                 if (equalityComparer.Equals(reference, linkIndex))
593                 {
594                     continue;
595                 }
596                 links.Update(reference, newLink, links.GetTarget(reference));
597             }
598             for (var i = (long)referencesAsSourceCount; i < references.Length; i++)
599             {
600                 var reference = references[i];
601                 if (equalityComparer.Equals(reference, linkIndex))
602                 {
603                     continue;
604                 }
605                 links.Update(reference, links.GetSource(reference), newLink);
606             }
607             ArrayPool.Free(references);
608         }
609     }
610     links.Delete(linkIndex);
611     return newLink;
612 }

```

```

609 }
610 }

./Incrementers/FrequencyIncrementer.cs
1  \texttt{
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Incrementers
6  {
7      public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>,
8          ↳ IIncrementer<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↳ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _frequencyMarker;
14         private readonly TLink _unaryOne;
15         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
16
17         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink
18             ↳ unaryOne, IIncrementer<TLink> unaryNumberIncrementer)
19             : base(links)
20         {
21             _frequencyMarker = frequencyMarker;
22             _unaryOne = unaryOne;
23             _unaryNumberIncrementer = unaryNumberIncrementer;
24         }
25
26         public TLink Increment(TLink frequency)
27         {
28             if (_equalityComparer.Equals(frequency, default))
29             {
30                 return Links.GetOrCreate(_unaryOne, _frequencyMarker);
31             }
32             var source = Links.GetSource(frequency);
33             var incrementedSource = _unaryNumberIncrementer.Increment(source);
34             return Links.GetOrCreate(incrementedSource, _frequencyMarker);
35         }
36     }
37 }

```

./Incrementers/LinkFrequencyIncrementer.cs

```

1  \texttt{
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Incrementers
6  {
7      public class LinkFrequencyIncrementer<TLink> : LinksOperatorBase<TLink>,
8          ↳ IIncrementer<ILink<TLink>>
9      {
10         private readonly ISpecificPropertyOperator<TLink, TLink>
11             ↳ _frequencyPropertyOperator;
12         private readonly IIncrementer<TLink> _frequencyIncrementer;
13
14         public LinkFrequencyIncrementer(ILinks<TLink> links,
15             ↳ ISpecificPropertyOperator<TLink, TLink> frequencyPropertyOperator,
16             ↳ IIncrementer<TLink> frequencyIncrementer)
17             : base(links)
18         {

```

```

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

```

./Incrementers/UnaryNumberIncrementer.cs

```

1  \texttt{
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Incrementers
6  {
7      public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>,
8          ↳ IIncrementer<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↳ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _unaryOne;
14
15         public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links)
16             ↳ => _unaryOne = unaryOne;
17
18         public TLink Increment(TLink unaryNumber)
19         {
20             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
21             {
22                 return Links.GetOrCreate(_unaryOne, _unaryOne);
23             }
24             var source = Links.GetSource(unaryNumber);
25             var target = Links.GetTarget(unaryNumber);
26             if (_equalityComparer.Equals(source, target))
27             {
28                 return Links.GetOrCreate(unaryNumber, _unaryOne);
29             }
30             else
31             {
32                 return Links.GetOrCreate(source, Increment(target));
33             }
34         }
35     }
36 }

```

```

33     }
34 }

./ISynchronizedLinks.cs
1  namespace Platform.Data.Constants;
2  namespace Platform.Data.Doublets
3  {
4      public interface ISynchronizedLinks<TLink> : ISynchronizedLinks<TLink,
5          ↳ ILinks<TLink>, LinksCombinedConstants<TLink, TLink, int>>, ILinks<TLink>
6      {
7      }
8  }
9
10 }
```

```

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

```

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

```

95     get
96     {
97         Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
98             ↪ nameof(index));
99         if (index == _constants.IndexPart)
100         {
101             return Index;
102         }
103         if (index == _constants.SourcePart)
104         {
105             return Source;
106         }
107         if (index == _constants.TargetPart)
108         {
109             return Target;
110         }
111         throw new NotSupportedException(); // Impossible path due to
112             ↪ Ensure.ArgumentInRange
113     }
114     set => throw new NotSupportedException();
115 }
116
117 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
118
119 public IEnumerator<TLink> GetEnumerator()
120 {
121     yield return Index;
122     yield return Source;
123     yield return Target;
124 }
125
126 public void Add(TLink item) => throw new NotSupportedException();
127
128 public void Clear() => throw new NotSupportedException();
129
130 public bool Contains(TLink item) => IndexOf(item) >= 0;
131
132 public void CopyTo(TLink[] array, int arrayIndex)
133 {
134     Ensure.Always.ArgumentNotNull(array, nameof(array));
135     Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length -
136         ↪ 1), nameof(arrayIndex));
137     if (arrayIndex + Length > array.Length)
138     {
139         throw new InvalidOperationException();
140     }
141     array[arrayIndex++] = Index;
142     array[arrayIndex++] = Source;
143     array[arrayIndex] = Target;
144 }
145
146 public bool Remove(TLink item) =>
147     ↪ Throw.A.NotSupportedExceptionAndReturn<bool>();
148
149 public int IndexOf(TLink item)
150 {
151     if (_equalityComparer.Equals(Index, item))
152     {
153         return _constants.IndexPart;
154     }
155     if (_equalityComparer.Equals(Source, item))
156     {
157         return _constants.SourcePart;
158     }
159     if (_equalityComparer.Equals(Target, item))
160     {
161         return _constants.TargetPart;
162     }
163     return -1;
164 }
165
166 #endregion
167 }
168 }
169

```

```

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 }
168 }
169

```

./LinkExtensions.cs

```

1  [texttt{
2  namespace Platform.Data.Doublets
3  {
4      public static class LinkExtensions
5      {
6          public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
7              ↪ Point<TLink>.IsFullPoint(link);
8          public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
9              ↪ Point<TLink>.IsPartialPoint(link);
10     }
11 }

```

./LinksOperatorBase.cs

```

1  [texttt{
2  namespace Platform.Data.Doublets
3  {
4      public abstract class LinksOperatorBase<TLink>
5      {
6          protected readonly ILinks<TLink> Links;
7          protected LinksOperatorBase(ILinks<TLink> links) => Links = links;
8      }
9  }
10 }

```

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

```

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

```

```

17 {assembly: System.Reflection.AssemblyVersionAttribute("0.0.1.0")}
18 }

```

./PropertyOperators/DefaultLinkPropertyOperator.cs

```

1  [texttt{
2  using System.Linq;
3  using System.Collections.Generic;
4  using Platform.Interfaces;
5
6  namespace Platform.Data.Doublets.PropertyOperators
7  {
8      public class DefaultLinkPropertyOperator<TLink> : LinksOperatorBase<TLink>,
9          ↳ IPropertyOperator<TLink, TLink, TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↳ EqualityComparer<TLink>.Default;
13
14         public DefaultLinkPropertyOperator(ILinks<TLink> links) : base(links)
15         {
16         }
17
18         public TLink GetValue(TLink @object, TLink property)
19         {
20             var objectProperty = Links.SearchOrDefault(@object, property);
21             if (_equalityComparer.Equals(objectProperty, default))
22             {
23                 return default;
24             }
25             var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
26             if (valueLink == null)
27             {
28                 return default;
29             }
30             var value = Links.GetTarget(valueLink[Links.Constants.IndexPart]);
31             return value;
32         }
33
34         public void SetValue(TLink @object, TLink property, TLink value)
35         {
36             var objectProperty = Links.GetOrCreate(@object, property);
37             Links.DeleteMany(Links.All(Links.Constants.Any, objectProperty).Select(link =>
38                 ↳ link[Links.Constants.IndexPart]).ToList());
39             Links.GetOrCreate(objectProperty, value);
40         }
41     }
42 }

```

./PropertyOperators/FrequencyPropertyOperator.cs

```

1  [texttt{
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.PropertyOperators
6  {
7      public class FrequencyPropertyOperator<TLink> : LinksOperatorBase<TLink>,
8          ↳ ISpecificPropertyOperator<TLink, TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↳ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _frequencyPropertyMarker;

```

```

12 private readonly TLink _frequencyMarker;
13
14 public FrequencyPropertyOperator(ILinks<TLink> links, TLink
15     ↳ frequencyPropertyMarker, TLink frequencyMarker) : base(links)
16 {
17     _frequencyPropertyMarker = frequencyPropertyMarker;
18     _frequencyMarker = frequencyMarker;
19 }
20
21 public TLink Get(TLink link)
22 {
23     var property = Links.SearchOrDefault(link, _frequencyPropertyMarker);
24     var container = GetContainer(property);
25     var frequency = GetFrequency(container);
26     return frequency;
27 }
28
29 private TLink GetContainer(TLink property)
30 {
31     var frequencyContainer = default(TLink);
32     if (_equalityComparer.Equals(property, default))
33     {
34         return frequencyContainer;
35     }
36     Links.Each(candidate =>
37     {
38         var candidateTarget = Links.GetTarget(candidate);
39         var frequencyTarget = Links.GetTarget(candidateTarget);
40         if (_equalityComparer.Equals(frequencyTarget, _frequencyMarker))
41         {
42             frequencyContainer = Links.GetIndex(candidate);
43             return Links.Constants.Break;
44         }
45         return Links.Constants.Continue;
46     }, Links.Constants.Any, property, Links.Constants.Any);
47     return frequencyContainer;
48 }
49
50 private TLink GetFrequency(TLink container) =>
51     ↳ _equalityComparer.Equals(container, default) ? default :
52     ↳ Links.GetTarget(container);
53
54 public void Set(TLink link, TLink frequency)
55 {
56     var property = Links.GetOrCreate(link, _frequencyPropertyMarker);
57     var container = GetContainer(property);
58     if (_equalityComparer.Equals(container, default))
59     {
60         Links.GetOrCreate(property, frequency);
61     }
62     else
63     {
64         Links.Update(container, property, frequency);
65     }
66 }

```

./ResizableDirectMemory/ResizableDirectMemoryLinks.cs

```

1  [texttt{
2  using System;

```

```

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

```

```

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

```

```

97 public static readonly int ReservedLinksOffset =
98     ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(ReservedLinks)).ToInt32();
99 public static readonly int FreeLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
100     ↳ nameof(FreeLinks)).ToInt32();
101 public static readonly int FirstFreeLinkOffset =
102     ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstFreeLink)).ToInt32();
103 public static readonly int FirstAsSourceOffset =
104     ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsSource)).ToInt32();
105 public static readonly int FirstAsTargetOffset =
106     ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsTarget)).ToInt32();
107 public static readonly int LastFreeLinkOffset =
108     ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(LastFreeLink)).ToInt32();
109
110 public TLink AllocatedLinks;
111 public TLink ReservedLinks;
112 public TLink FreeLinks;
113 public TLink FirstFreeLink;
114 public TLink FirstAsSource;
115 public TLink FirstAsTarget;
116 public TLink LastFreeLink;
117 public TLink Reserved8;
118
119 [MethodImpl(MethodImplOptions.AggressiveInlining)]
120 public static TLink GetAllocatedLinks(IntPtr pointer) => (pointer +
121     ↳ AllocatedLinksOffset).GetValue<TLink>();
122 [MethodImpl(MethodImplOptions.AggressiveInlining)]
123 public static TLink GetReservedLinks(IntPtr pointer) => (pointer +
124     ↳ ReservedLinksOffset).GetValue<TLink>();
125 [MethodImpl(MethodImplOptions.AggressiveInlining)]
126 public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
127     ↳ FreeLinksOffset).GetValue<TLink>();
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
130     ↳ FirstFreeLinkOffset).GetValue<TLink>();
131 [MethodImpl(MethodImplOptions.AggressiveInlining)]
132 public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
133     ↳ FirstAsSourceOffset).GetValue<TLink>();
134 [MethodImpl(MethodImplOptions.AggressiveInlining)]
135 public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
136     ↳ FirstAsTargetOffset).GetValue<TLink>();
137 [MethodImpl(MethodImplOptions.AggressiveInlining)]
138 public static TLink GetLastFreeLink(IntPtr pointer) => (pointer +
139     ↳ LastFreeLinkOffset).GetValue<TLink>();
140
141 [MethodImpl(MethodImplOptions.AggressiveInlining)]
142 public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) => pointer +
143     ↳ FirstAsSourceOffset;
144 [MethodImpl(MethodImplOptions.AggressiveInlining)]
145 public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
146     ↳ FirstAsTargetOffset;
147
148 [MethodImpl(MethodImplOptions.AggressiveInlining)]
149 public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
150     ↳ AllocatedLinksOffset).SetValue(value);
151 [MethodImpl(MethodImplOptions.AggressiveInlining)]
152 public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
153     ↳ ReservedLinksOffset).SetValue(value);
154 [MethodImpl(MethodImplOptions.AggressiveInlining)]
155 public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
156     ↳ FreeLinksOffset).SetValue(value);
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158
159 public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
160     ↳ FirstFreeLinkOffset).SetValue(value);
161 [MethodImpl(MethodImplOptions.AggressiveInlining)]
162 public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
163     ↳ FirstAsSourceOffset).SetValue(value);
164 [MethodImpl(MethodImplOptions.AggressiveInlining)]
165 public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
166     ↳ FirstAsTargetOffset).SetValue(value);
167 [MethodImpl(MethodImplOptions.AggressiveInlining)]
168 public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
169     ↳ LastFreeLinkOffset).SetValue(value);
170 }
171
172 private readonly long _memoryReservationStep;
173
174 private readonly IResizableDirectMemory _memory;
175 private IntPtr _header;
176 private IntPtr _links;
177
178 private LinksTargetsTreeMethods _targetsTreeMethods;
179 private LinksSourcesTreeMethods _sourcesTreeMethods;
180
181 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь
182     ↳ удалённой, нужно использовать не список а дерево, так как так можно
183     ↳ быстрее проверить на наличие связи внутри
184 private UnusedLinksListMethods _unusedLinksListMethods;
185
186 /// <summary>
187 /// Возвращает общее число связей находящихся в хранилище.
188 /// </summary>
189 private TLink Total => Subtract(LinksHeader.GetAllocatedLinks(_header),
190     ↳ LinksHeader.GetFreeLinks(_header));
191
192 public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
193
194 public ResizableDirectMemoryLinks(string address)
195     : this(address, DefaultLinksSizeStep)
196 {
197 }
198
199 /// <summary>
200 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с
201     ↳ указанным минимальным шагом расширения базы данных.
202 /// </summary>
203 /// <param name="address">Полный путь к файлу базы данных.</param>
204 /// <param name="memoryReservationStep">Минимальный шаг расширения базы
205     ↳ данных в байтах.</param>
206 public ResizableDirectMemoryLinks(string address, long memoryReservationStep)
207     : this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
208     ↳ memoryReservationStep)
209 {
210 }
211
212 public ResizableDirectMemoryLinks(IResizableDirectMemory memory)
213     : this(memory, DefaultLinksSizeStep)
214 {
215 }
216
217 public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
218     ↳ memoryReservationStep)
219 {
220     Constants = Default<LinksCombinedConstants<TLink, TLink, int>>.Instance;
221 }

```

```

191     _memory = memory;
192     _memoryReservationStep = memoryReservationStep;
193     if (memory.ReservedCapacity < memoryReservationStep)
194     {
195         memory.ReservedCapacity = memoryReservationStep;
196     }
197     SetPointers(_memory);
198     // Гарантия корректности _memory.UsedCapacity относительно
199     ↪ _header->AllocatedLinks
200     _memory.UsedCapacity =
201     ↪ ((long)(Integer<TLink>)LinksHeader.GetAllocatedLinks(_header) *
202     ↪ LinkSizeInBytes) + LinkHeaderSizeInBytes;
203     // Гарантия корректности _header->ReservedLinks относительно
204     ↪ _memory.ReservedCapacity
205     LinksHeader.SetReservedLinks(_header,
206     ↪ (Integer<TLink>)((_memory.ReservedCapacity - LinkHeaderSizeInBytes) /
207     ↪ LinkSizeInBytes));
208 }
209
210 [MethodImpl(MethodImplOptions.AggressiveInlining)]
211 public TLink Count(IList<TLink> restrictions)
212 {
213     // Если нет ограничений, тогда возвращаем общее число связей находящихся в
214     ↪ хранилище.
215     if (restrictions.Count == 0)
216     {
217         return Total;
218     }
219     if (restrictions.Count == 1)
220     {
221         var index = restrictions[Constants.IndexPart];
222         if (_equalityComparer.Equals(index, Constants.Any))
223         {
224             return Total;
225         }
226         return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
227     }
228     if (restrictions.Count == 2)
229     {
230         var index = restrictions[Constants.IndexPart];
231         var value = restrictions[1];
232         if (_equalityComparer.Equals(index, Constants.Any))
233         {
234             if (_equalityComparer.Equals(value, Constants.Any))
235             {
236                 return Total; // Any - как отсутствие ограничения
237             }
238             return Add(_sourcesTreeMethods.CalculateReferences(value),
239             ↪ _targetsTreeMethods.CalculateReferences(value));
240         }
241         else
242         {
243             if (!Exists(index))
244             {
245                 return Integer<TLink>.Zero;
246             }
247             if (_equalityComparer.Equals(value, Constants.Any))
248             {
249                 return Integer<TLink>.One;
250             }
251             var storedLinkValue = GetLinkUnsafe(index);
252             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||

```

```

245         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
246         {
247             return Integer<TLink>.One;
248         }
249         return Integer<TLink>.Zero;
250     }
251 }
252 if (restrictions.Count == 3)
253 {
254     var index = restrictions[Constants.IndexPart];
255     var source = restrictions[Constants.SourcePart];
256     var target = restrictions[Constants.TargetPart];
257
258     if (_equalityComparer.Equals(index, Constants.Any))
259     {
260         if (_equalityComparer.Equals(source, Constants.Any) &&
261         ↪ _equalityComparer.Equals(target, Constants.Any))
262         {
263             return Total;
264         }
265         else if (_equalityComparer.Equals(source, Constants.Any))
266         {
267             return _targetsTreeMethods.CalculateReferences(target);
268         }
269         else if (_equalityComparer.Equals(target, Constants.Any))
270         {
271             return _sourcesTreeMethods.CalculateReferences(source);
272         }
273         else //if(source != Any && target != Any)
274         {
275             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
276             var link = _sourcesTreeMethods.Search(source, target);
277             return _equalityComparer.Equals(link, Constants.Null) ?
278             ↪ Integer<TLink>.Zero : Integer<TLink>.One;
279         }
280     }
281     else
282     {
283         if (!Exists(index))
284         {
285             return Integer<TLink>.Zero;
286         }
287         if (_equalityComparer.Equals(source, Constants.Any) &&
288         ↪ _equalityComparer.Equals(target, Constants.Any))
289         {
290             return Integer<TLink>.One;
291         }
292         var storedLinkValue = GetLinkUnsafe(index);
293         if (!_equalityComparer.Equals(source, Constants.Any) &&
294         ↪ !_equalityComparer.Equals(target, Constants.Any))
295         {
296             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source)
297             ↪ &&
298             ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
299             {
300                 return Integer<TLink>.One;
301             }
302             return Integer<TLink>.Zero;
303         }
304         var value = default(TLink);
305         if (_equalityComparer.Equals(source, Constants.Any))

```



```

301     {
302         value = target;
303     }
304     if (_equalityComparer.Equals(target, Constants.Any))
305     {
306         value = source;
307     }
308     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
309         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
310     {
311         return Integer<TLink>.One;
312     }
313     return Integer<TLink>.Zero;
314 }
315 }
316 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
317 }
318
319 [MethodImpl(MethodImplOptions.AggressiveInlining)]
320 public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
321 {
322     if (restrictions.Count == 0)
323     {
324         for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
    ↳ (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
    ↳ Increment(link))
325         {
326             if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
    ↳ Constants.Break))
327             {
328                 return Constants.Break;
329             }
330         }
331         return Constants.Continue;
332     }
333     if (restrictions.Count == 1)
334     {
335         var index = restrictions[Constants.IndexPart];
336         if (_equalityComparer.Equals(index, Constants.Any))
337         {
338             return Each(handler, ArrayPool<TLink>.Empty);
339         }
340         if (!Exists(index))
341         {
342             return Constants.Continue;
343         }
344         return handler(GetLinkStruct(index));
345     }
346     if (restrictions.Count == 2)
347     {
348         var index = restrictions[Constants.IndexPart];
349         var value = restrictions[1];
350         if (_equalityComparer.Equals(index, Constants.Any))
351         {
352             if (_equalityComparer.Equals(value, Constants.Any))
353             {
354                 return Each(handler, ArrayPool<TLink>.Empty);
355             }
356             if (_equalityComparer.Equals(Each(handler, new[] { index, value,
    ↳ Constants.Any }), Constants.Break))
357

```

```

358     {
359         return Constants.Break;
360     }
361     return Each(handler, new[] { index, Constants.Any, value });
362 }
363 else
364 {
365     if (!Exists(index))
366     {
367         return Constants.Continue;
368     }
369     if (_equalityComparer.Equals(value, Constants.Any))
370     {
371         return handler(GetLinkStruct(index));
372     }
373     var storedLinkValue = GetLinkUnsafe(index);
374     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
375         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
376     {
377         return handler(GetLinkStruct(index));
378     }
379     return Constants.Continue;
380 }
381 }
382 if (restrictions.Count == 3)
383 {
384     var index = restrictions[Constants.IndexPart];
385     var source = restrictions[Constants.SourcePart];
386     var target = restrictions[Constants.TargetPart];
387     if (_equalityComparer.Equals(index, Constants.Any))
388     {
389         if (_equalityComparer.Equals(source, Constants.Any) &&
    ↳ _equalityComparer.Equals(target, Constants.Any))
390         {
391             return Each(handler, ArrayPool<TLink>.Empty);
392         }
393         else if (_equalityComparer.Equals(source, Constants.Any))
394         {
395             return _targetsTreeMethods.EachReference(target, handler);
396         }
397         else if (_equalityComparer.Equals(target, Constants.Any))
398         {
399             return _sourcesTreeMethods.EachReference(source, handler);
400         }
401         else //if(source != Any && target != Any)
402         {
403             var link = _sourcesTreeMethods.Search(source, target);
404             return _equalityComparer.Equals(link, Constants.Null) ?
    ↳ Constants.Continue : handler(GetLinkStruct(link));
405         }
406     }
407     else
408     {
409         if (!Exists(index))
410         {
411             return Constants.Continue;
412         }
413         if (_equalityComparer.Equals(source, Constants.Any) &&
    ↳ _equalityComparer.Equals(target, Constants.Any))
414         {
415             return handler(GetLinkStruct(index));
416         }
417     }
418 }

```

```

416     }
417     var storedLinkValue = GetLinkUnsafe(index);
418     if (!_equalityComparer.Equals(source, Constants.Any) &&
419         ↪ !_equalityComparer.Equals(target, Constants.Any))
420     {
421         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source)
422             ↪ &&
423             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
424         {
425             return handler(GetLinkStruct(index));
426         }
427         return Constants.Continue;
428     }
429     var value = default(TLink);
430     if (_equalityComparer.Equals(source, Constants.Any))
431     {
432         value = target;
433     }
434     if (_equalityComparer.Equals(target, Constants.Any))
435     {
436         value = source;
437     }
438     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
439         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
440     {
441         return handler(GetLinkStruct(index));
442     }
443     return Constants.Continue;
444 }
445 }
446
447 /// <remarks>
448 /// TODO: Возможно можно перемещать значения, если указан индекс, но
449 ↪ значение существует в другом месте (но не в менеджере памяти, а в логике
450 ↪ Links)
451 /// </remarks>
452 [MethodImpl(MethodImplOptions.AggressiveInlining)]
453 public TLink Update(ICollection<TLink> values)
454 {
455     var linkIndex = values[Constants.IndexPart];
456     var link = GetLinkUnsafe(linkIndex);
457     // Будет корректно работать только в том случае, если пространство
458     ↪ выделенной связи предварительно заполнено нулями
459     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
460     {
461         _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
462             ↪ linkIndex);
463     }
464     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
465     {
466         _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
467             ↪ linkIndex);
468     }
469     Link.SetSource(link, values[Constants.SourcePart]);
470     Link.SetTarget(link, values[Constants.TargetPart]);
471     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
472     {
473

```

```

468         _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer(_header),
469             ↪ linkIndex);
470     }
471     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
472     {
473         _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTargetPointer(_header),
474             ↪ linkIndex);
475     }
476     return linkIndex;
477 }
478
479 [MethodImpl(MethodImplOptions.AggressiveInlining)]
480 public Link<TLink> GetLinkStruct(TLink linkIndex)
481 {
482     var link = GetLinkUnsafe(linkIndex);
483     return new Link<TLink>(linkIndex, Link.GetSource(link), Link.GetTarget(link));
484 }
485
486 [MethodImpl(MethodImplOptions.AggressiveInlining)]
487 private IntPtr GetLinkUnsafe(TLink linkIndex) =>
488     ↪ _links.GetElement(LinkSizeInBytes, linkIndex);
489
490 /// <remarks>
491 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не
492 ↪ заполняет пространство
493 /// </remarks>
494 public TLink Create()
495 {
496     var freeLink = LinksHeader.GetFirstFreeLink(_header);
497     if (!_equalityComparer.Equals(freeLink, Constants.Null))
498     {
499         _unusedLinksList Methods.Detach(freeLink);
500     }
501     else
502     {
503         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
504             ↪ Constants.MaxPossibleIndex) > 0)
505         {
506             throw new LinksLimitReachedException(((Integer<TLink>)Constants.MaxP
507                 ↪ ossibleIndex);
508         }
509         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
510             ↪ Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
511         {
512             memory.ReservedCapacity += _memoryReservationStep;
513             SetPointers(_memory);
514             LinksHeader.SetReservedLinks(_header,
515                 ↪ (Integer<TLink>)(_memory.ReservedCapacity / LinkSizeInBytes));
516         }
517         LinksHeader.SetAllocatedLinks(_header,
518             ↪ Increment(LinksHeader.GetAllocatedLinks(_header)));
519         memory.UsedCapacity += LinkSizeInBytes;
520         freeLink = LinksHeader.GetAllocatedLinks(_header);
521     }
522     return freeLink;
523 }
524
525 public void Delete(TLink link)
526 {
527     if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)
528     {
529         _unusedLinksList Methods.AttachAsFirst(link);
530     }
531 }

```



```

35     protected override void SetPrevious(TLink element, TLink previous) =>
36         ↪ ( _links.GetElement(LinkSizeInBytes, element) +
37         ↪ Link.SourceOffset).SetValue(previous);
38
39     protected override void SetNext(TLink element, TLink next) =>
40         ↪ ( _links.GetElement(LinkSizeInBytes, element) +
41         ↪ Link.TargetOffset).SetValue(next);
42
43     protected override void SetSize(TLink size) => ( _header +
44         ↪ LinksHeader.FreeLinksOffset).SetValue(size);
45 }
46 }
47 }

```

./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs

```

1  namespace ResizableDirectMemoryLinks{
2  using System;
3  using System.Text;
4  using System.Collections.Generic;
5  using System.Runtime.CompilerServices;
6  using Platform.Numbers;
7  using Platform.Unsafe;
8  using Platform.Collections.Methods.Trees;
9  using Platform.Data.Constants;
10
11 namespace Platform.Data.Doublets.ResizableDirectMemory
12 {
13     partial class ResizableDirectMemoryLinks<TLink>
14     {
15         private abstract class LinksTreeMethodsBase :
16             ↪ SizedAndThreadedAVLBalancedTreeMethods<TLink>
17         {
18             private readonly ResizableDirectMemoryLinks<TLink> _memory;
19             private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
20             protected readonly IntPtr Links;
21             protected readonly IntPtr Header;
22
23             protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
24             {
25                 Links = memory._links;
26                 Header = memory._header;
27                 _memory = memory;
28                 _constants = memory.Constants;
29             }
30
31             [MethodImpl(MethodImplOptions.AggressiveInlining)]
32             protected abstract TLink GetTreeRoot();
33
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             protected abstract TLink GetBasePartValue(TLink link);
36
37             public TLink this[TLink index]
38             {
39                 get
40                 {
41                     var root = GetTreeRoot();
42                     if (GreaterOrEqualThan(index, GetSize(root)))
43                     {
44                         return GetZero();
45                     }
46                     while (!EqualToZero(root))

```

```

47         var left = GetLeftOrDefault(root);
48         var leftSize = GetSizeOrZero(left);
49         if (LessThan(index, leftSize))
50         {
51             root = left;
52             continue;
53         }
54         if (IsEquals(index, leftSize))
55         {
56             return root;
57         }
58         root = GetRightOrDefault(root);
59         index = Subtract(index, Increment(leftSize));
60     }
61     return GetZero(); // TODO: Impossible situation exception (only if tree
62         ↪ structure broken)
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         else
94         {
95             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
96             root = GetRightOrDefault(root);
97         }
98     }
99     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
100 }
101
102 public TLink EachReference(TLink link, Func<ILink<TLink>, TLink> handler)
103 {
104     var root = GetTreeRoot();
105     if (EqualToZero(root))
106     {
107

```

```

108         return _constants.Continue;
109     }
110     TLink first = GetZero(), current = root;
111     while (!EqualToZero(current))
112     {
113         var @base = GetBasePartValue(current);
114         if (GreaterOrEqualThan(@base, link))
115         {
116             if (IsEquals(@base, link))
117             {
118                 first = current;
119             }
120             current = GetLeftOrDefault(current);
121         }
122         else
123         {
124             current = GetRightOrDefault(current);
125         }
126     }
127     if (!EqualToZero(first))
128     {
129         current = first;
130         while (true)
131         {
132             if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
133             {
134                 return _constants.Break;
135             }
136             current = GetNext(current);
137             if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
138             {
139                 break;
140             }
141         }
142     }
143     return _constants.Continue;
144 }
145
146 protected override void PrintNodeValue(TLink node, StringBuilder sb)
147 {
148     sb.Append(' ');
149     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
150         ↪ Link.SourceOffset).GetValue<TLink>());
151     sb.Append('-');
152     sb.Append('>');
153     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
154         ↪ Link.TargetOffset).GetValue<TLink>());
155 }
156
157 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
158 {
159     public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
160         : base(memory)
161     {
162     }
163
164     protected override IntPtr GetLeftPointer(TLink node) =>
165         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset;
166
167     protected override IntPtr GetRightPointer(TLink node) =>
168         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset;

```

```

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

```

```

209     var modified = BitwiseHelpers.PartialWrite(previousValue,
210         ↪ (TLink)(Integer<TLink>)value, 3, 1);
211     (Links.GetElement(LinkSizeInBytes, node) +
212     ↪ Link.SizeAsSourceOffset).SetValue(modified);
213 }
214 protected override sbyte GetBalance(TLink node)
215 {
216     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
217     ↪ Link.SizeAsSourceOffset).GetValue<TLink>());
218     var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue,
219     ↪ 0, 3);
220     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
221     ↪ 124 : value & 3);
222     return unpackedValue;
223 }
224 protected override void SetBalance(TLink node, sbyte value)
225 {
226     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
227     ↪ Link.SizeAsSourceOffset).GetValue<TLink>());
228     var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) |
229     ↪ value & 3);
230     var modified = BitwiseHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
231     (Links.GetElement(LinkSizeInBytes, node) +
232     ↪ Link.SizeAsSourceOffset).SetValue(modified);
233 }
234 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
235 {
236     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
237     ↪ Link.SourceOffset).GetValue<TLink>());
238     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
239     ↪ Link.SourceOffset).GetValue<TLink>());
240     return LessThan(firstSource, secondSource) ||
241     (IsEquals(firstSource, secondSource) &&
242     ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
243     ↪ Link.TargetOffset).GetValue<TLink>(),
244     ↪ (Links.GetElement(LinkSizeInBytes, second) +
245     ↪ Link.TargetOffset).GetValue<TLink>()));
246 }
247 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
248 {
249     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
250     ↪ Link.SourceOffset).GetValue<TLink>());
251     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
252     ↪ Link.SourceOffset).GetValue<TLink>());
253     return GreaterThan(firstSource, secondSource) ||
254     (IsEquals(firstSource, secondSource) &&
255     ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
256     ↪ Link.TargetOffset).GetValue<TLink>(),
257     ↪ (Links.GetElement(LinkSizeInBytes, second) +
258     ↪ Link.TargetOffset).GetValue<TLink>()));
259 }
260 protected override TLink GetTreeRoot() => (Header +
261     ↪ LinksHeader.FirstAsSourceOffset).GetValue<TLink>());

```

```

247 protected override TLink GetBasePartValue(TLink link) =>
248     ↪ (Links.GetElement(LinkSizeInBytes, link) +
249     ↪ Link.SourceOffset).GetValue<TLink>());
250
251 /// <summary>
252 /// Выполняет поиск и возвращает индекс связи с указанными Source
253     ↪ (началом) и Target (концом)
254     ↪ по дереву (индексу) связей, отсортированному по Source, а затем по Target.
255     ↪ </summary>
256     ↪ <param name="source">Индекс связи, которая является началом на
257     ↪ искомой связи.</param>
258     ↪ <param name="target">Индекс связи, которая является концом на искомой
259     ↪ связи.</param>
260     ↪ <returns>Индекс искомой связи.</returns>
261 public TLink Search(TLink source, TLink target)
262 {
263     var root = GetTreeRoot();
264     while (!EqualToZero(root))
265     {
266         var rootSource = (Links.GetElement(LinkSizeInBytes, root) +
267         ↪ Link.SourceOffset).GetValue<TLink>());
268         var rootTarget = (Links.GetElement(LinkSizeInBytes, root) +
269         ↪ Link.TargetOffset).GetValue<TLink>());
270         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
271         ↪ node.Key < root.Key
272         {
273             root = GetLeftOrDefault(root);
274         }
275         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
276         ↪ // node.Key > root.Key
277         {
278             root = GetRightOrDefault(root);
279         }
280         else // node.Key == root.Key
281         {
282             return root;
283         }
284     }
285     return GetZero();
286 }
287
288 [MethodImpl(MethodImplOptions.AggressiveInlining)]
289 private bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget, TLink
290     ↪ secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
291     ↪ (IsEquals(firstSource, secondSource) && LessThan(firstTarget,
292     ↪ secondTarget));
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
296     ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource,
297     ↪ secondSource) || (IsEquals(firstSource, secondSource) &&
298     ↪ GreaterThan(firstTarget, secondTarget));
299 }
300 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
301 {
302     public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
303     : base(memory)
304     {
305     }
306 }

```

```

293 protected override IntPtr GetLeftPointer(TLink node) =>
↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset;
294
295 protected override IntPtr GetRightPointer(TLink node) =>
↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset;
296
297 protected override TLink GetLeftValue(TLink node) =>
↪ (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.LeftAsTargetOffset).GetValue<TLink>();
298
299 protected override TLink GetRightValue(TLink node) =>
↪ (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.RightAsTargetOffset).GetValue<TLink>();
300
301 protected override TLink GetSize(TLink node)
302 {
303     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.SizeAsTargetOffset).GetValue<TLink>());
304     return BitwiseHelpers.PartialRead(previousValue, 5, -5);
305 }
306
307 protected override void SetLeft(TLink node, TLink left) =>
↪ (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.LeftAsTargetOffset).SetValue(left);
308
309 protected override void SetRight(TLink node, TLink right) =>
↪ (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.RightAsTargetOffset).SetValue(right);
310
311 protected override void SetSize(TLink node, TLink size)
312 {
313     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.SizeAsTargetOffset).GetValue<TLink>());
314     (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsTargetOffset).SetValue(
↪ e(BitwiseHelpers.PartialWrite(previousValue, size, 5,
↪ -5)));
315 }
316
317 protected override bool GetLeftIsChild(TLink node)
318 {
319     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.SizeAsTargetOffset).GetValue<TLink>());
320     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 4, 1);
321 }
322
323 protected override void SetLeftIsChild(TLink node, bool value)
324 {
325     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.SizeAsTargetOffset).GetValue<TLink>());
326     var modified = BitwiseHelpers.PartialWrite(previousValue,
↪ (TLink)(Integer<TLink>)value, 4, 1);
327     (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.SizeAsTargetOffset).SetValue(modified);
328 }
329
330 protected override bool GetRightIsChild(TLink node)
331 {
332     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.SizeAsTargetOffset).GetValue<TLink>());
333     return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 3, 1);
334 }

```

```

335
336 protected override void SetRightIsChild(TLink node, bool value)
337 {
338     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.SizeAsTargetOffset).GetValue<TLink>());
339     var modified = BitwiseHelpers.PartialWrite(previousValue,
↪ (TLink)(Integer<TLink>)value, 3, 1);
340     (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.SizeAsTargetOffset).SetValue(modified);
341 }
342
343 protected override sbyte GetBalance(TLink node)
344 {
345     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.SizeAsTargetOffset).GetValue<TLink>());
346     var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue,
↪ 0, 3);
347     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
↪ 124 : value & 3);
348     return unpackedValue;
349 }
350
351 protected override void SetBalance(TLink node, sbyte value)
352 {
353     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.SizeAsTargetOffset).GetValue<TLink>());
354     var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) |
↪ value & 3);
355     var modified = BitwiseHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
356     (Links.GetElement(LinkSizeInBytes, node) +
↪ Link.SizeAsTargetOffset).SetValue(modified);
357 }
358
359 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
360 {
361     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
↪ Link.TargetOffset).GetValue<TLink>());
362     var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
↪ Link.TargetOffset).GetValue<TLink>());
363     return LessThan(firstTarget, secondTarget) ||
364         (IsEquals(firstTarget, secondTarget) &&
↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
↪ Link.SourceOffset).GetValue<TLink>(),
↪ (Links.GetElement(LinkSizeInBytes, second) +
↪ Link.SourceOffset).GetValue<TLink>()));
365 }
366
367 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
368 {
369     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
↪ Link.TargetOffset).GetValue<TLink>());
370     var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
↪ Link.TargetOffset).GetValue<TLink>());
371     return GreaterThan(firstTarget, secondTarget) ||
372         (IsEquals(firstTarget, secondTarget) &&
↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
↪ Link.SourceOffset).GetValue<TLink>(),
↪ (Links.GetElement(LinkSizeInBytes, second) +
↪ Link.SourceOffset).GetValue<TLink>()));
373 }

```

```

374         protected override TLink GetTreeRoot() => (Header +
375             ↳ LinksHeader.FirstAsTargetOffset).GetValue<TLink>());
376
377     protected override TLink GetBasePartValue(TLink link) =>
378         ↳ (Links.GetElement(LinkSizeInBytes, link) +
379             ↳ Link.TargetOffset).GetValue<TLink>());
380     }
381 }

```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs

```

1  namespace Platform.Data.Doublets.ResizableDirectMemory
2  {
3      using System;
4      using System.Collections.Generic;
5      using System.Runtime.CompilerServices;
6      using Platform.Disposables;
7      using Platform.Collections.Arrays;
8      using Platform.Helpers.Singletons;
9      using Platform.Memory;
10     using Platform.Data.Exceptions;
11     using Platform.Data.Constants;
12
13     // #define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
14     #pragma warning disable 0649
15     #pragma warning disable 169
16
17     // ReSharper disable BuiltInTypeReferenceStyle
18
19     namespace Platform.Data.Doublets.ResizableDirectMemory
20     {
21         using id = UInt64;
22
23         public unsafe partial class UInt64ResizableDirectMemoryLinks : DisposableBase,
24             ↳ ILinks<id>
25         {
26             /// <summary>Возвращает размер одной связи в байтах.</summary>
27             /// <remarks>
28             ///     Используется только во вне класса, не рекомендуется использовать внутри.
29             ///     Так как во вне не обязательно будет доступен unsafe C#.
30             /// </remarks>
31             public static readonly int LinkSizeInBytes = sizeof(Link);
32
33             public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
34
35             private struct Link
36             {
37                 public id Source;
38                 public id Target;
39                 public id LeftAsSource;
40                 public id RightAsSource;
41                 public id SizeAsSource;
42                 public id LeftAsTarget;
43                 public id RightAsTarget;
44                 public id SizeAsTarget;
45             }
46
47             private struct LinksHeader
48             {
49                 public id AllocatedLinks;
50                 public id ReservedLinks;
51                 public id FreeLinks;
52             }
53         }
54     }
55 }

```

```

51     public id FirstFreeLink;
52     public id FirstAsSource;
53     public id FirstAsTarget;
54     public id LastFreeLink;
55     public id Reserved8;
56 }
57
58 private readonly long _memoryReservationStep;
59
60 private readonly IResizableDirectMemory _memory;
61 private LinksHeader* _header;
62 private Link* _links;
63
64 private LinksTargetsTreeMethods _targetsTreeMethods;
65 private LinksSourcesTreeMethods _sourcesTreeMethods;
66
67 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь
68 ↳ удалённой, нужно использовать не список а дерево, так как так можно
69 ↳ быстрее проверить на наличие связи внутри
70 private UnusedLinksListMethods _unusedLinksListMethods;
71
72 /// <summary>
73 ///     Возвращает общее число связей находящихся в хранилище.
74 /// </summary>
75 private id Total => _header->AllocatedLinks - _header->FreeLinks;
76
77 // TODO: Дать возможность переопределять в конструкторе
78 public LinksCombinedConstants<id, id, int> Constants { get; }
79
80 public UInt64ResizableDirectMemoryLinks(string address) : this(address,
81     ↳ DefaultLinksSizeStep) { }
82
83 /// <summary>
84 ///     Создаёт экземпляр базы данных Links в файле по указанному адресу, с
85 ↳ указанным минимальным шагом расширения базы данных.
86 /// </summary>
87 /// <param name="address">Полный путь к файлу базы данных.</param>
88 /// <param name="memoryReservationStep">Минимальный шаг расширения базы
89 ↳ данных в байтах.</param>
90 public UInt64ResizableDirectMemoryLinks(string address, long
91     ↳ memoryReservationStep) : this(new FileMappedResizableDirectMemory(address,
92     ↳ memoryReservationStep), memoryReservationStep) { }
93
94 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) :
95     ↳ this(memory, DefaultLinksSizeStep) { }
96
97 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
98     ↳ memoryReservationStep)
99 {
100     Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
101     _memory = memory;
102     _memoryReservationStep = memoryReservationStep;
103     if (memory.ReservedCapacity < memoryReservationStep)
104     {
105         memory.ReservedCapacity = memoryReservationStep;
106     }
107     SetPointers(_memory);
108     // Гарантия корректности _memory.UsedCapacity относительно
109     ↳ _header->AllocatedLinks
110     _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) +
111     ↳ sizeof(LinksHeader);
112     // Гарантия корректности _header->ReservedLinks относительно
113     ↳ _memory.ReservedCapacity

```



```

102     _header->ReservedLinks = (id)((_memory.ReservedCapacity -
    ↪     sizeof(LinksHeader)) / sizeof(Link));
103 }
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 public id Count(IList<id> restrictions)
107 {
108     // Если нет ограничений, тогда возвращаем общее число связей находящихся в
    ↪     хранилище.
109     if (restrictions.Count == 0)
110     {
111         return Total;
112     }
113     if (restrictions.Count == 1)
114     {
115         var index = restrictions[Constants.IndexPart];
116         if (index == Constants.Any)
117         {
118             return Total;
119         }
120         return Exists(index) ? 1UL : 0UL;
121     }
122     if (restrictions.Count == 2)
123     {
124         var index = restrictions[Constants.IndexPart];
125         var value = restrictions[1];
126         if (index == Constants.Any)
127         {
128             if (value == Constants.Any)
129             {
130                 return Total; // Any - как отсутствие ограничения
131             }
132             return _sourcesTreeMethods.CalculateReferences(value)
133                 + _targetsTreeMethods.CalculateReferences(value);
134         }
135         else
136         {
137             if (!Exists(index))
138             {
139                 return 0;
140             }
141             if (value == Constants.Any)
142             {
143                 return 1;
144             }
145             var storedLinkValue = GetLinkUnsafe(index);
146             if (storedLinkValue->Source == value ||
147                 storedLinkValue->Target == value)
148             {
149                 return 1;
150             }
151             return 0;
152         }
153     }
154     if (restrictions.Count == 3)
155     {
156         var index = restrictions[Constants.IndexPart];
157         var source = restrictions[Constants.SourcePart];
158         var target = restrictions[Constants.TargetPart];
159         if (index == Constants.Any)
160         {
161             if (source == Constants.Any && target == Constants.Any)

```

```

162     {
163         return Total;
164     }
165     else if (source == Constants.Any)
166     {
167         return _targetsTreeMethods.CalculateReferences(target);
168     }
169     else if (target == Constants.Any)
170     {
171         return _sourcesTreeMethods.CalculateReferences(source);
172     }
173     else //if(source != Any && target != Any)
174     {
175         // ЭКВИВАЛЕНТ Exists(source, target) => Count(Any, source, target) > 0
176         var link = _sourcesTreeMethods.Search(source, target);
177         return link == Constants.Null ? 0UL : 1UL;
178     }
179 }
180 else
181 {
182     if (!Exists(index))
183     {
184         return 0;
185     }
186     if (source == Constants.Any && target == Constants.Any)
187     {
188         return 1;
189     }
190     var storedLinkValue = GetLinkUnsafe(index);
191     if (source != Constants.Any && target != Constants.Any)
192     {
193         if (storedLinkValue->Source == source &&
194             storedLinkValue->Target == target)
195         {
196             return 1;
197         }
198         return 0;
199     }
200     var value = default(id);
201     if (source == Constants.Any)
202     {
203         value = target;
204     }
205     if (target == Constants.Any)
206     {
207         value = source;
208     }
209     if (storedLinkValue->Source == value ||
210         storedLinkValue->Target == value)
211     {
212         return 1;
213     }
214     return 0;
215 }
216 }
217
218 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↪     поддерживаются.");
219 }
220
221 [MethodImpl(MethodImplOptions.AggressiveInlining)]
222 public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
223 {

```

```

223 if (restrictions.Count == 0)
224 {
225     for (id link = 1; link <= _header->AllocatedLinks; link++)
226     {
227         if (Exists(link))
228         {
229             if (handler(GetLinkStruct(link)) == Constants.Break)
230             {
231                 return Constants.Break;
232             }
233         }
234     }
235     return Constants.Continue;
236 }
237 if (restrictions.Count == 1)
238 {
239     var index = restrictions[Constants.IndexPart];
240     if (index == Constants.Any)
241     {
242         return Each(handler, ArrayPool<ulong>.Empty);
243     }
244     if (!Exists(index))
245     {
246         return Constants.Continue;
247     }
248     return handler(GetLinkStruct(index));
249 }
250 if (restrictions.Count == 2)
251 {
252     var index = restrictions[Constants.IndexPart];
253     var value = restrictions[1];
254     if (index == Constants.Any)
255     {
256         if (value == Constants.Any)
257         {
258             return Each(handler, ArrayPool<ulong>.Empty);
259         }
260         if (Each(handler, new[] { index, value, Constants.Any }) == Constants.Break)
261         {
262             return Constants.Break;
263         }
264         return Each(handler, new[] { index, Constants.Any, value });
265     }
266     else
267     {
268         if (!Exists(index))
269         {
270             return Constants.Continue;
271         }
272         if (value == Constants.Any)
273         {
274             return handler(GetLinkStruct(index));
275         }
276         var storedLinkValue = GetLinkUnsafe(index);
277         if (storedLinkValue->Source == value ||
278             storedLinkValue->Target == value)
279         {
280             return handler(GetLinkStruct(index));
281         }
282         return Constants.Continue;
283     }
284 }

```

```

285 if (restrictions.Count == 3)
286 {
287     var index = restrictions[Constants.IndexPart];
288     var source = restrictions[Constants.SourcePart];
289     var target = restrictions[Constants.TargetPart];
290     if (index == Constants.Any)
291     {
292         if (source == Constants.Any && target == Constants.Any)
293         {
294             return Each(handler, ArrayPool<ulong>.Empty);
295         }
296         else if (source == Constants.Any)
297         {
298             return _targetsTreeMethods.EachReference(target, handler);
299         }
300         else if (target == Constants.Any)
301         {
302             return _sourcesTreeMethods.EachReference(source, handler);
303         }
304         else //if(source != Any && target != Any)
305         {
306             var link = _sourcesTreeMethods.Search(source, target);
307             return link == Constants.Null ? Constants.Continue :
308                 ↪ handler(GetLinkStruct(link));
309         }
310     }
311     else
312     {
313         if (!Exists(index))
314         {
315             return Constants.Continue;
316         }
317         if (source == Constants.Any && target == Constants.Any)
318         {
319             return handler(GetLinkStruct(index));
320         }
321         var storedLinkValue = GetLinkUnsafe(index);
322         if (source != Constants.Any && target != Constants.Any)
323         {
324             if (storedLinkValue->Source == source &&
325                 storedLinkValue->Target == target)
326             {
327                 return handler(GetLinkStruct(index));
328             }
329             return Constants.Continue;
330         }
331         var value = default(id);
332         if (source == Constants.Any)
333         {
334             value = target;
335         }
336         if (target == Constants.Any)
337         {
338             value = source;
339         }
340         if (storedLinkValue->Source == value ||
341             storedLinkValue->Target == value)
342         {
343             return handler(GetLinkStruct(index));
344         }
345     }
346     return Constants.Continue;
347 }

```

```

345     }
346 }
347 throw new NotSupportedException("Другие размеры и способы ограничений не
↳ поддерживаются.");
348 }
349
350 /// <remarks>
351 /// TODO: Возможно можно перемещать значения, если указан индекс, но
↳ значение существует в другом месте (но не в менеджере памяти, а в логике
↳ Links)
352 /// </remarks>
353 [MethodImpl(MethodImplOptions.AggressiveInlining)]
354 public id Update(ICollection<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 предварительно заполнено нулями
359     if (link->Source != Constants.Null)
360     {
361         _sourcesTreeMethods.Detach(new IntPtr(&_header->FirstAsSource), linkIndex);
362     }
363     if (link->Target != Constants.Null)
364     {
365         _targetsTreeMethods.Detach(new IntPtr(&_header->FirstAsTarget), linkIndex);
366     }
367 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
368     var leftTreeSize = _sourcesTreeMethods.GetSize(new
↳ IntPtr(&_header->FirstAsSource));
369     var rightTreeSize = _targetsTreeMethods.GetSize(new
↳ IntPtr(&_header->FirstAsTarget));
370     if (leftTreeSize != rightTreeSize)
371     {
372         throw new Exception("One of the trees is broken.");
373     }
374 #endif
375     link->Source = values[Constants.SourcePart];
376     link->Target = values[Constants.TargetPart];
377     if (link->Source != Constants.Null)
378     {
379         _sourcesTreeMethods.Attach(new IntPtr(&_header->FirstAsSource), linkIndex);
380     }
381     if (link->Target != Constants.Null)
382     {
383         _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
384     }
385 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
386     leftTreeSize = _sourcesTreeMethods.GetSize(new
↳ IntPtr(&_header->FirstAsSource));
387     rightTreeSize = _targetsTreeMethods.GetSize(new
↳ IntPtr(&_header->FirstAsTarget));
388     if (leftTreeSize != rightTreeSize)
389     {
390         throw new Exception("One of the trees is broken.");
391     }
392 #endif
393     return linkIndex;
394 }
395
396 [MethodImpl(MethodImplOptions.AggressiveInlining)]
397 private IList<id> GetLinkStruct(id linkIndex)

```

```

398 {
399     var link = GetLinkUnsafe(linkIndex);
400     return new UInt64Link(linkIndex, link->Source, link->Target);
401 }
402
403 [MethodImpl(MethodImplOptions.AggressiveInlining)]
404 private Link* GetLinkUnsafe(id linkIndex) => &_amp;links[linkIndex];
405
406 /// <remarks>
407 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не
↳ заполняет пространство
408 /// </remarks>
409 public id Create()
410 {
411     var freeLink = _header->FirstFreeLink;
412     if (freeLink != Constants.Null)
413     {
414         _unusedLinksListMethods.Detach(freeLink);
415     }
416     else
417     {
418         if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
419         {
420             throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
421         }
422         if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
423         {
424             _memory.ReservedCapacity += _memory.ReservationStep;
425             SetPointers(_memory);
426             _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
427         }
428         _header->AllocatedLinks++;
429         _memory.UsedCapacity += sizeof(Link);
430         freeLink = _header->AllocatedLinks;
431     }
432     return freeLink;
433 }
434
435 public void Delete(id link)
436 {
437     if (link < _header->AllocatedLinks)
438     {
439         _unusedLinksListMethods.AttachAsFirst(link);
440     }
441     else if (link == _header->AllocatedLinks)
442     {
443         _header->AllocatedLinks--;
444         _memory.UsedCapacity -= sizeof(Link);
445         // Убираем все связи, находящиеся в списке свободных в конце файла, до
↳ тех пор, пока не дойдём до первой существующей связи
446         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
447         while (_header->AllocatedLinks > 0 &&
↳ IsUnusedLink(_header->AllocatedLinks))
448         {
449             _unusedLinksListMethods.Detach(_header->AllocatedLinks);
450             _header->AllocatedLinks--;
451             _memory.UsedCapacity -= sizeof(Link);
452         }
453     }
454 }
455
456 /// <remarks>

```

```

457  /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том
458  /// случае, если адрес реально менялся
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(IResizableDirectMemory memory)
465  {
466      if (memory == null)
467      {
468          header = null;
469          _links = null;
470          _unusedLinksListMethods = null;
471          _targetsTreeMethods = null;
472          _unusedLinksListMethods = null;
473      }
474      else
475      {
476          header = (LinksHeader*)(void*)memory.Pointer;
477          _links = (Link*)(void*)memory.Pointer;
478          _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
479          _targetsTreeMethods = new LinksTargetsTreeMethods(this);
480          _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
481      }
482
483      [MethodImpl(MethodImplOptions.AggressiveInlining)]
484      private bool Exists(id link) => link >= Constants.MinPossibleIndex && link <=
485      ↪ _header->AllocatedLinks && !IsUnusedLink(link);
486
487      [MethodImpl(MethodImplOptions.AggressiveInlining)]
488      private bool IsUnusedLink(id link) => header->FirstFreeLink == link
489      || (_links[link].SizeAsSource == Constants.Null &&
490      ↪ _links[link].Source != Constants.Null);
491
492      #region Disposable
493
494      protected override bool AllowMultipleDisposeCalls => true;
495
496      protected override void DisposeCore(bool manual, bool wasDisposed)
497      {
498          if (!wasDisposed)
499          {
500              SetPointers(null);
501              Disposable.TryDispose(_memory);
502          }
503      }
504      #endregion
505  }
506  }

```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs

```

1  [texttt{
2  using Platform.Collections.Methods.Lists;
3
4  namespace Platform.Data.Doublets.ResizableDirectMemory
5  {
6      unsafe partial class UInt64ResizableDirectMemoryLinks
7      {
8          private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<ulong>

```

```

9  {
10
11      private readonly Link* _links;
12      private readonly LinksHeader* _header;
13
14      public UnusedLinksListMethods(Link* links, LinksHeader* header)
15      {
16          _links = links;
17          _header = header;
18      }
19
20      protected override ulong GetFirst() => _header->FirstFreeLink;
21
22      protected override ulong GetLast() => _header->LastFreeLink;
23
24      protected override ulong GetPrevious(ulong element) => _links[element].Source;
25
26      protected override ulong GetNext(ulong element) => _links[element].Target;
27
28      protected override ulong GetSize() => _header->FreeLinks;
29
30      protected override void SetFirst(ulong element) => _header->FirstFreeLink =
31      ↪ element;
32
33      protected override void SetLast(ulong element) => _header->LastFreeLink =
34      ↪ element;
35
36      protected override void SetPrevious(ulong element, ulong previous) =>
37      ↪ _links[element].Source = previous;
38
39      protected override void SetNext(ulong element, ulong next) =>
40      ↪ _links[element].Target = next;
41
42      protected override void SetSize(ulong size) => _header->FreeLinks = size;
43  }
44  }
45  }

```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs

```

1  [texttt{
2  using System;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using Platform.Collections.Methods.Trees;
7  using Platform.Data.Constants;
8
9  namespace Platform.Data.Doublets.ResizableDirectMemory
10 {
11     unsafe partial class UInt64ResizableDirectMemoryLinks
12     {
13         private abstract class LinksTreeMethodsBase :
14         ↪ SizedAndThreadedAVLBalancedTreeMethods<ulong>
15         {
16             private readonly UInt64ResizableDirectMemoryLinks _memory;
17             private readonly LinksCombinedConstants<ulong, ulong, int> _constants;
18             protected readonly Link* Links;
19             protected readonly LinksHeader* Header;
20
21             protected LinksTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory)
22             {
23                 Links = memory._links;
24                 Header = memory._header;

```

```

24     _memory = memory;
25     _constants = memory.Constants;
26 }
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 protected abstract ulong GetTreeRoot();
30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 protected abstract ulong GetBasePartValue(ulong link);
33
34 public ulong this[ulong index]
35 {
36     get
37     {
38         var root = GetTreeRoot();
39         if (index >= GetSize(root))
40         {
41             return 0;
42         }
43         while (root != 0)
44         {
45             var left = GetLeftOrDefault(root);
46             var leftSize = GetSizeOrZero(left);
47             if (index < leftSize)
48             {
49                 root = left;
50                 continue;
51             }
52             if (index == leftSize)
53             {
54                 return root;
55             }
56             root = GetRightOrDefault(root);
57             index -= leftSize + 1;
58         }
59         return 0; // TODO: Impossible situation exception (only if tree structure
60                 ↪ broken)
61     }
62 }
63
64 // TODO: Return indices range instead of references count
65 public ulong CalculateReferences(ulong link)
66 {
67     var root = GetTreeRoot();
68     var total = GetSize(root);
69     var totalRightIgnore = 0UL;
70     while (root != 0)
71     {
72         var @base = GetBasePartValue(root);
73         if (@base <= link)
74         {
75             root = GetRightOrDefault(root);
76         }
77         else
78         {
79             totalRightIgnore += GetRightSize(root) + 1;
80             root = GetLeftOrDefault(root);
81         }
82     }
83     root = GetTreeRoot();
84     var totalLeftIgnore = 0UL;
85     while (root != 0)
86     {

```

```

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

```

```

148         sb.Append('>');
149         sb.Append(Links[node].Target);
150     }
151 }
152
153 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
154 {
155     public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)
156     : base(memory)
157     {
158     }
159
160     protected override IntPtr GetLeftPointer(ulong node) => new
161     ↪ IntPtr(&Links[node].Left.AsSource);
162
163     protected override IntPtr GetRightPointer(ulong node) => new
164     ↪ IntPtr(&Links[node].Right.AsSource);
165
166     protected override ulong GetLeftValue(ulong node) => Links[node].Left.AsSource;
167
168     protected override ulong GetRightValue(ulong node) => Links[node].Right.AsSource;
169
170     protected override ulong GetSize(ulong node)
171     {
172         var previousValue = Links[node].Size.AsSource;
173         //return MathHelpers.PartialRead(previousValue, 5, -5);
174         return (previousValue & 4294967264) >> 5;
175     }
176
177     protected override void SetLeft(ulong node, ulong left) =>
178     ↪ Links[node].Left.AsSource = left;
179
180     protected override void SetRight(ulong node, ulong right) =>
181     ↪ Links[node].Right.AsSource = right;
182
183     protected override void SetSize(ulong node, ulong size)
184     {
185         var previousValue = Links[node].Size.AsSource;
186         //var modified = MathHelpers.PartialWrite(previousValue, size, 5, -5);
187         var modified = (previousValue & 31) | ((size & 134217727) << 5);
188         Links[node].Size.AsSource = modified;
189     }
190
191     protected override bool GetLeftIsChild(ulong node)
192     {
193         var previousValue = Links[node].Size.AsSource;
194         //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
195         return (previousValue & 16) >> 4 == 1UL;
196     }
197
198     protected override void SetLeftIsChild(ulong node, bool value)
199     {
200         var previousValue = Links[node].Size.AsSource;
201         //var modified = MathHelpers.PartialWrite(previousValue,
202         ↪ (ulong)(Integer)value, 4, 1);
203         var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
204         Links[node].Size.AsSource = modified;
205     }
206
207     protected override bool GetRightIsChild(ulong node)
208     {
209         var previousValue = Links[node].Size.AsSource;
210         //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);

```

```

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

```

```

261         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
262             ↪ node.Key < root.Key
263         {
264             root = GetLeftOrDefault(root);
265         }
266         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
267             ↪ // node.Key > root.Key
268         {
269             root = GetRightOrDefault(root);
270         }
271         else // node.Key == root.Key
272         {
273             return root;
274         }
275     }
276     return 0;
277 }
278 [MethodImpl(MethodImplOptions.AggressiveInlining)]
279 private static bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
280     ↪ ulong secondSource, ulong secondTarget)
281     => firstSource < secondSource || (firstSource == secondSource && firstTarget
282     ↪ < secondTarget);
283
284 [MethodImpl(MethodImplOptions.AggressiveInlining)]
285 private static bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
286     ↪ ulong secondSource, ulong secondTarget)
287     => firstSource > secondSource || (firstSource == secondSource && firstTarget
288     ↪ > secondTarget);
289
290 [MethodImpl(MethodImplOptions.AggressiveInlining)]
291 protected override void ClearNode(ulong node)
292 {
293     Links[node].LeftAsSource = 0UL;
294     Links[node].RightAsSource = 0UL;
295     Links[node].SizeAsSource = 0UL;
296 }
297
298 [MethodImpl(MethodImplOptions.AggressiveInlining)]
299 protected override ulong GetZero() => 0UL;
300
301 [MethodImpl(MethodImplOptions.AggressiveInlining)]
302 protected override ulong GetOne() => 1UL;
303
304 [MethodImpl(MethodImplOptions.AggressiveInlining)]
305 protected override ulong GetTwo() => 2UL;
306
307 [MethodImpl(MethodImplOptions.AggressiveInlining)]
308 protected override bool ValueEqualToZero(IntPtr pointer) =>
309     ↪ *(ulong*)pointer.ToPointer() == 0UL;
310
311 [MethodImpl(MethodImplOptions.AggressiveInlining)]
312 protected override bool EqualToZero(ulong value) => value == 0UL;
313
314 [MethodImpl(MethodImplOptions.AggressiveInlining)]
315 protected override bool GreaterThan(ulong first, ulong second) => first > second;
316

```

```

317 [MethodImpl(MethodImplOptions.AggressiveInlining)]
318 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first
319     ↪ >= second;
320
321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
322 protected override bool GreaterOrEqualThanZero(ulong value) => true; // value
323     ↪ >= 0 is always true for ulong
324
325 [MethodImpl(MethodImplOptions.AggressiveInlining)]
326 protected override bool LessOrEqualThanZero(ulong value) => value == 0; //
327     ↪ value is always >= 0 for ulong
328
329 [MethodImpl(MethodImplOptions.AggressiveInlining)]
330 protected override bool LessOrEqualThan(ulong first, ulong second) => first <=
331     ↪ second;
332
333 [MethodImpl(MethodImplOptions.AggressiveInlining)]
334 protected override bool LessThanZero(ulong value) => false; // value < 0 is always
335     ↪ false for ulong
336
337 [MethodImpl(MethodImplOptions.AggressiveInlining)]
338 protected override bool LessThan(ulong first, ulong second) => first < second;
339
340 [MethodImpl(MethodImplOptions.AggressiveInlining)]
341 protected override ulong Increment(ulong value) => ++value;
342
343 [MethodImpl(MethodImplOptions.AggressiveInlining)]
344 protected override ulong Decrement(ulong value) => --value;
345
346 [MethodImpl(MethodImplOptions.AggressiveInlining)]
347 protected override ulong Add(ulong first, ulong second) => first + second;
348
349 [MethodImpl(MethodImplOptions.AggressiveInlining)]
350 protected override ulong Subtract(ulong first, ulong second) => first - second;
351 }
352
353 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
354 {
355     public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
356         : base(memory)
357     {
358     }
359
360     //protected override IntPtr GetLeft(ulong node) => new
361     ↪ IntPtr(&Links[node].LeftAsTarget);
362
363     //protected override IntPtr GetRight(ulong node) => new
364     ↪ IntPtr(&Links[node].RightAsTarget);
365
366     //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
367
368     //protected override void SetLeft(ulong node, ulong left) =>
369     ↪ Links[node].LeftAsTarget = left;
370
371     //protected override void SetRight(ulong node, ulong right) =>
372     ↪ Links[node].RightAsTarget = right;
373
374     //protected override void SetSize(ulong node, ulong size) =>
375     ↪ Links[node].SizeAsTarget = size;
376
377     protected override IntPtr GetLeftPointer(ulong node) => new
378     ↪ IntPtr(&Links[node].LeftAsTarget);
379

```

```

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

```

```

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

```

./Sequences/Converters/BalancedVariantConverter.cs

```

1  [texttt{
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets.Sequences.Converters
5  {
6      public class BalancedVariantConverter<TLink> :
    ↳ LinksListToSequenceConverterBase<TLink>
7      {
8          public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }

```



```

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

```

./Sequences/Converters/CompressingConverter.cs

```

1  [texttt{
2  using System;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Interfaces;
6  using Platform.Collections;
7  using Platform.Helpers.Singletons;
8  using Platform.Numbers;
9  using Platform.Data.Constants;
10 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
11
12 namespace Platform.Data.Doublets.Sequences.Converters
13 {

```

```

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

```

```

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

```

```

121     {
122         sequence[i] = copy[i].Element;
123     }
124 }
125 return sequence;
126 }
127
128 /// <remarks>
129 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
130 /// </remarks>
131 private int ReplaceDoublets(HalfDoublet[] copy)
132 {
133     var oldLength = copy.Length;
134     var newLength = copy.Length;
135     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
136     {
137         var maxDoubletSource = _maxDoublet.Source;
138         var maxDoubletTarget = _maxDoublet.Target;
139         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
140         {
141             _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource,
142             ↪ maxDoubletTarget);
143         }
144         var maxDoubletReplacementLink = _maxDoubletData.Link;
145         oldLength--;
146         var oldLengthMinusTwo = oldLength - 1;
147         // Substitute all usages
148         int w = 0, r = 0; // (r == read, w == write)
149         for (; r < oldLength; r++)
150         {
151             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
152             ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
153             {
154                 if (r > 0)
155                 {
156                     var previous = copy[w - 1].Element;
157                     copy[w - 1].DoubletData.DecrementFrequency();
158                     copy[w - 1].DoubletData =
159                     ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
160                     ↪ maxDoubletReplacementLink);
161                 }
162                 if (r < oldLengthMinusTwo)
163                 {
164                     var next = copy[r + 2].Element;
165                     copy[r + 1].DoubletData.DecrementFrequency();
166                     copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(
167                     ↪ y(maxDoubletReplacementLink,
168                     ↪ next);
169                 }
170                 copy[w++].Element = maxDoubletReplacementLink;
171                 r++;
172                 newLength--;
173             }
174             else
175             {
176                 copy[w++] = copy[r];
177             }
178         }
179     }
180     if (w < newLength)
181     {
182         copy[w] = copy[r];

```

```

176     }
177     oldLength = newLength;
178     ResetMaxDoublet();
179     UpdateMaxDoublet(copy, newLength);
180 }
181 return newLength;
182 }
183
184 [MethodImpl(MethodImplOptions.AggressiveInlining)]
185 private void ResetMaxDoublet()
186 {
187     _maxDoublet = new Doublet<TLink>();
188     _maxDoubletData = new LinkFrequency<TLink>();
189 }
190
191 [MethodImpl(MethodImplOptions.AggressiveInlining)]
192 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
193 {
194     Doublet<TLink> doublet = default;
195     for (var i = 1; i < length; i++)
196     {
197         doublet.Source = copy[i - 1].Element;
198         doublet.Target = copy[i].Element;
199         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
200     }
201 }
202
203 [MethodImpl(MethodImplOptions.AggressiveInlining)]
204 private void UpdateMaxDoublet(ref Doublet<TLink> doublet,
205     ↪ LinkFrequency<TLink> data)
206 {
207     var frequency = data.Frequency;
208     var maxFrequency = _maxDoubletData.Frequency;
209     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
210     ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives
211     ↪ better compression string data (and gives collisions quickly) */
212     ↪ _maxDoublet.Source + _maxDoublet.Target)))
213     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
214     ↪ (_comparer.Compare(maxFrequency, frequency) < 0 ||
215     ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
216     ↪ _comparer.Compare(ArithmeticHelpers.Add(doublet.Source,
217     ↪ doublet.Target), ArithmeticHelpers.Add(_maxDoublet.Source,
218     ↪ _maxDoublet.Target)) > 0))) /* gives better stability and better
219     ↪ compression on sequent data and even on runder numbers data (but gives
220     ↪ collisions anyway) */
221     {
222         _maxDoublet = doublet;
223         _maxDoubletData = data;
224     }
225 }
226 }
227 }
228 }

```

./Sequences/Converters/LinksListToSequenceConverterBase.cs

```

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

```

```

7      public abstract class LinksListToSequenceConverterBase<TLink> :
8      ↪ IConverter<IList<TLink>, TLink>
9      {
10         protected readonly ILinks<TLink> Links;
11         public LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
12         public abstract TLink Convert(IList<TLink> source);
13     }
14 }

```

./Sequences/Converters/OptimalVariantConverter.cs

```

1  [texttt{
2  using System.Collections.Generic;
3  using System.Linq;
4  using Platform.Interfaces;
5
6  namespace Platform.Data.Doublets.Sequences.Converters
7  {
8      public class OptimalVariantConverter<TLink> :
9      ↪ LinksListToSequenceConverterBase<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12         ↪ EqualityComparer<TLink>.Default;
13         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
14
15         private readonly IConverter<IList<TLink>>
16         ↪ _sequenceToItsLocalElementLevelsConverter;
17
18         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
19         ↪ sequenceToItsLocalElementLevelsConverter) : base(links)
20         => _sequenceToItsLocalElementLevelsConverter =
21         ↪ sequenceToItsLocalElementLevelsConverter;
22
23         public override TLink Convert(IList<TLink> sequence)
24         {
25             var length = sequence.Count;
26             if (length == 1)
27             {
28                 return sequence[0];
29             }
30             var links = Links;
31             if (length == 2)
32             {
33                 return links.GetOrCreate(sequence[0], sequence[1]);
34             }
35             sequence = sequence.ToArray();
36             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
37             while (length > 2)
38             {
39                 var levelRepeat = 1;
40                 var currentLevel = levels[0];
41                 var previousLevel = levels[0];
42                 var skipOnce = false;
43                 var w = 0;
44                 for (var i = 1; i < length; i++)
45                 {
46                     if (_equalityComparer.Equals(currentLevel, levels[i]))

```

```

47     sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
48     var newLevel = i >= length - 1 ?
49         GetPreviousLowerThanCurrentOrCurrent(previousLevel,
50             ↪ 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 else
68 {
69     currentLevel = levels[i];
70     levelRepeat = 1;
71     if (skipOnce)
72     {
73         skipOnce = false;
74     }
75     else
76     {
77         sequence[w] = sequence[i - 1];
78         levels[w] = levels[i - 1];
79         previousLevel = levels[w];
80         w++;
81     }
82     if (i == length - 1)
83     {
84         sequence[w] = sequence[i];
85         levels[w] = levels[i];
86         w++;
87     }
88 }
89 length = w;
90 }
91 return links.GetOrCreate(sequence[0], sequence[1]);
92 }
93
94 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink
95     ↪ previous, TLink current, TLink next)
96 {
97     return _comparer.Compare(previous, next) > 0
98     ? _comparer.Compare(previous, current) < 0 ? previous : current
99     : _comparer.Compare(next, current) < 0 ? next : current;
100 }
101
102 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink
103     ↪ next) => _comparer.Compare(next, current) < 0 ? next : current;
104
105 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink
106     ↪ current) => _comparer.Compare(previous, current) < 0 ? previous : current;
107 }

```

```

105 }
106 }

```

./Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs

```

1  \texttt{
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Sequences.Converters
6  {
7      public class SequenceToItsLocalElementLevelsConverter<TLink> :
8          ↪ LinksOperatorBase<TLink>, IConverter<IList<TLink>>
9      {
10         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11         private readonly IConverter<Doublet<TLink>, TLink>
12             ↪ _linkToItsFrequencyToNumberConverter;
13         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
14             ↪ IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter)
15             ↪ : base(links) => _linkToItsFrequencyToNumberConveter =
16             ↪ linkToItsFrequencyToNumberConveter;
17         public IList<TLink> Convert(IList<TLink> sequence)
18         {
19             var levels = new TLink[sequence.Count];
20             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
21             for (var i = 1; i < sequence.Count - 1; i++)
22             {
23                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
24                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
25                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
26             }
27             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
28                 ↪ sequence[sequence.Count - 1]);
29             return levels;
30         }
31
32         public TLink GetFrequencyNumber(TLink source, TLink target) =>
33             ↪ _linkToItsFrequencyToNumberConverter.Convert(new Doublet<TLink>(source,
34                 ↪ target));
35     }
36 }

```

./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs

```

1  \texttt{
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
5  {
6      public class DefaultSequenceElementCreteriaMat cher<TLink> :
7          ↪ LinksOperatorBase<TLink>, ICreteriaMat cher<TLink>
8      {
9         public DefaultSequenceElementCreteriaMat cher(ILinks<TLink> links) : base(links) { }
10         public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);
11     }
12 }

```

./Sequences/CreteriaMatchers/MarkedSequenceCreteriaMatcher.cs

```

1  \texttt{
2  using System.Collections.Generic;

```

```

3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Sequences.CriteriaMatchers
6  {
7      public class MarkedSequenceCriteriaMatcher<TLink> : ICriteriaMatcher<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ⇨ EqualityComparer<TLink>.Default;
11
12          private readonly ILinks<TLink> _links;
13          private readonly TLink _sequenceMarkerLink;
14
15          public MarkedSequenceCriteriaMatcher(ILinks<TLink> links, TLink
16             ⇨ sequenceMarkerLink)
17          {
18              _links = links;
19              _sequenceMarkerLink = sequenceMarkerLink;
20          }
21
22          public bool IsMatched(TLink sequenceCandidate)
23             => _equalityComparer.Equals(_links.GetSource(sequenceCandidate),
24             ⇨ _sequenceMarkerLink)
25             || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
26             ⇨ sequenceCandidate), _links.Constants.Null);
27
28      }
29  }

```

./Sequences/DefaultSequenceAppender.cs

```

1  \texttt{
2  using System.Collections.Generic;
3  using Platform.Collections.Stacks;
4  using Platform.Data.Doublets.Sequences.HeightProviders;
5  using Platform.Data.Sequences;
6
7  namespace Platform.Data.Doublets.Sequences
8  {
9      public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
10         ⇨ ISequenceAppender<TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
13             ⇨ EqualityComparer<TLink>.Default;
14
15          private readonly IStack<TLink> _stack;
16          private readonly ISequenceHeightProvider<TLink> _heightProvider;
17
18          public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
19             ⇨ ISequenceHeightProvider<TLink> heightProvider)
20             : base(links)
21          {
22              _stack = stack;
23              _heightProvider = heightProvider;
24          }
25
26          public TLink Append(TLink sequence, TLink appendant)
27          {
28              var cursor = sequence;
29              while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
30              {
31                  var source = Links.GetSource(cursor);
32                  var target = Links.GetTarget(cursor);
33                  if (_equalityComparer.Equals(_heightProvider.Get(source),
34                     ⇨ _heightProvider.Get(target)))

```

```

31      {
32          break;
33      }
34      else
35      {
36          _stack.Push(source);
37          _cursor = target;
38      }
39  }
40  var left = cursor;
41  var right = appendant;
42  while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
43  {
44      right = Links.GetOrCreate(left, right);
45      left = cursor;
46  }
47  return Links.GetOrCreate(left, right);
48  }
49  }
50  }
51  }

```

./Sequences/DuplicateSegmentsCounter.cs

```

1  \texttt{
2  using System.Collections.Generic;
3  using System.Linq;
4  using Platform.Interfaces;
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public class DuplicateSegmentsCounter<TLink> : ICounter<int>
9      {
10         private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
11             ⇨ _duplicateFragmentsProvider;
12         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
13             ⇨ IList<TLink>>>> duplicateFragmentsProvider) =>
14             ⇨ _duplicateFragmentsProvider = duplicateFragmentsProvider;
15         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
16     }
17 }

```

./Sequences/DuplicateSegmentsProvider.cs

```

1  \texttt{
2  using System;
3  using System.Linq;
4  using System.Collections.Generic;
5  using Platform.Interfaces;
6  using Platform.Collections;
7  using Platform.Collections.Lists;
8  using Platform.Collections.Segments;
9  using Platform.Collections.Segments.Walkers;
10 using Platform.Helpers;
11 using Platform.Helpers.Singletons;
12 using Platform.Numbers;
13 using Platform.Data.Sequences;
14
15 namespace Platform.Data.Doublets.Sequences
16 {
17     public class DuplicateSegmentsProvider<TLink> :
18         ⇨ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
19         ⇨ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>

```

```

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

```

```

68         sequences.EachPart(sequenceElements.AddAndReturnTrue, linkIndex);
69         if (sequenceElements.Count > 2)
70         {
71             WalkAll(sequenceElements);
72         }
73     }
74     return _links.Constants.Continue;
75 });
76 var resultList = _groups.ToList();
77 var comparer = Default<ItemComparer>.Instance;
78 resultList.Sort(comparer);
79 #if DEBUG
80     foreach (var item in resultList)
81     {
82         PrintDuplicates(item);
83     }
84 #endif
85     return resultList;
86 }
87
88     protected override Segment<TLink> CreateSegment(IList<TLink> elements, int
89         ↳ offset, int length) => new Segment<TLink>(elements, offset, length);
90
91     protected override void OnDuplicateFound(Segment<TLink> segment)
92     {
93         var duplicates = CollectDuplicatesForSegment(segment);
94         if (duplicates.Count > 1)
95         {
96             _groups.Add(new KeyValuePair<IList<TLink>,
97                 ↳ IList<TLink>>(segment.ToArray(), duplicates));
98         }
99     }
100
101     private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
102     {
103         var duplicates = new List<TLink>();
104         var readAsElement = new HashSet<TLink>();
105         sequences.Each(sequence =>
106         {
107             duplicates.Add(sequence);
108             readAsElement.Add(sequence);
109             return true; // Continue
110         }, segment);
111         if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
112         {
113             return new List<TLink>();
114         }
115         foreach (var duplicate in duplicates)
116         {
117             var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
118             _visited.Set(duplicateBitIndex);
119         }
120         if (_sequences is Sequences sequencesExperiments)
121         {
122             var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences(
123                 ↳ es4((HashSet<ulong>)(object)readAsElement,
124                 ↳ (IList<ulong>)segment);
125             foreach (var partiallyMatchedSequence in partiallyMatched)
126             {
127                 TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
128                 duplicates.Add(sequenceIndex);
129             }
130         }

```

```

125     }
126 }
127 duplicates.Sort();
128 return duplicates;
129 }
130
131 private void PrintDuplicates(KeyValuePair<ILink<TLink>, ILink<TLink>>
    ↳ duplicatesItem)
132 {
133     if (!(_links is ILinks<ulong> ulongLinks))
134     {
135         return;
136     }
137     var duplicatesKey = duplicatesItem.Key;
138     var keyString = UnicodeMap.FromLinksToString((ILink<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
            => Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
            ↳ UnicodeMap.IsCharLink(link.Index) ?
            ↳ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) :
            ↳ sb.Append(link.Index));
145         Console.WriteLine(formattedSequenceStructure);
146         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
            ↳ ulongLinks);
147         Console.WriteLine(sequenceString);
148     }
149     Console.WriteLine();
150 }
151 }
152 }
153 }

```

./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs

```

1  \texttt{
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
6  {
7      public class FrequenciesCacheBasedLinkFrequencyIncrementer<TLink> :
8          ↳ IIncrementer<ILink<TLink>>
9      {
10         private readonly LinkFrequenciesCache<TLink> _cache;
11
12         public FrequenciesCacheBasedLinkFrequencyIncrementer(LinkFrequenciesCache<TL
13             ↳ ink> cache) => _cache =
14             ↳ cache;
15
16         /// <remarks>Sequence itseft is not changed, only frequency of its doublets is
17             ↳ incremented.</remarks>
18         public ILink<TLink> Increment(ILink<TLink> sequence)
19         {
20             _cache.IncrementFrequencies(sequence);
21             return sequence;
22         }
23     }
24 }
25 }

```

./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToItsFrequencyNumberConverter

```

1  \texttt{
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
5  {
6      public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
7          ↳ IConverter<Doublet<TLink>, TLink>
8      {
9         private readonly LinkFrequenciesCache<TLink> _cache;
10         public FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequencies
11             ↳ Cache<TLink> cache) => _cache =
12             ↳ cache;
13         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref
14             ↳ source).Frequency;
15     }
16 }

```

./Sequences/Frequencies/Cache/LinkFrequenciesCache.cs

```

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

```

```

36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
38 {
39     doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
40     return data;
41 }
42
43 public void IncrementFrequencies(ICollection<TLink> sequence)
44 {
45     for (var i = 1; i < sequence.Count; i++)
46     {
47         IncrementFrequency(sequence[i - 1], sequence[i]);
48     }
49 }
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
53 {
54     var doublet = new Doublet<TLink>(source, target);
55     return IncrementFrequency(ref doublet);
56 }
57
58 public void PrintFrequencies(ICollection<TLink> sequence)
59 {
60     for (var i = 1; i < sequence.Count; i++)
61     {
62         PrintFrequency(sequence[i - 1], sequence[i]);
63     }
64 }
65
66 public void PrintFrequency(TLink source, TLink target)
67 {
68     var number = GetFrequency(source, target).Frequency;
69     Console.WriteLine("{0},{1} - {2}", source, target, number);
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
74 {
75     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
76     {
77         data.IncrementFrequency();
78     }
79     else
80     {
81         var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
82         data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
83         if (!_equalityComparer.Equals(link, default))
84         {
85             data.Frequency = ArithmeticHelpers.Add(data.Frequency,
86                 ↪ _frequencyCounter.Count(link));
87         }
88         _doubletsCache.Add(doublet, data);
89     }
90     return data;
91 }
92
93 public void ValidateFrequencies()
94 {
95     foreach (var entry in _doubletsCache)
96     {
97         var value = entry.Value;
98         var linkIndex = value.Link;

```

```

98         if (!_equalityComparer.Equals(linkIndex, default))
99         {
100             var frequency = value.Frequency;
101             var count = _frequencyCounter.Count(linkIndex);
102             // TODO: Why `frequency` always greater than `count` by 1?
103             if (((_comparer.Compare(frequency, count) > 0) &&
104                 ↪ (_comparer.Compare(ArithmeticHelpers.Subtract(frequency, count),
105                     ↪ Integer<TLink>.One) > 0))
106                 || ((_comparer.Compare(count, frequency) > 0) &&
107                     ↪ (_comparer.Compare(ArithmeticHelpers.Subtract(count, frequency),
108                         ↪ Integer<TLink>.One) > 0)))
109             {
110                 throw new InvalidOperationException("Frequencies validation failed.");
111             }
112         }
113         // else
114         // {
115         //     if (value.Frequency > 0)
116         //     {
117         //         var frequency = value.Frequency;
118         //         linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
119         //         var count = _countLinkFrequency(linkIndex);
120         //         // if ((frequency > count && frequency - count > 1) || (count > frequency
121         //         //     ↪ && count - frequency > 1))
122         //         //     throw new Exception("Frequencies validation failed.");
123         //     }
124         // }
125     }

```

./Sequences/Frequencies/Cache/LinkFrequency.cs

```

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

```



```

25         public override string ToString() => $"F: {Frequency}, L: {Link}";
26     }
27 }
28 }
29 }

./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs
1  \texttt{
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
5  {
6      public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
7          ↪ SequenceSymbolFrequencyOneOffCounter<TLink>
8      {
9          private readonly ICriteriaMatcher<TLink> _markedSequenceMatcher;
10
11         public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
12             ↪ ICriteriaMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink
13             ↪ symbol)
14             : base(links, sequenceLink, symbol)
15             => _markedSequenceMatcher = markedSequenceMatcher;
16
17         public override TLink Count()
18         {
19             if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
20             {
21                 return default;
22             }
23             return base.Count();
24         }
25     }
26 }
27 }
28 }

```

```

./Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs
1  \texttt{
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Numbers;
5  using Platform.Data.Sequences;
6
7  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
8  {
9      public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
14
15         protected readonly ILinks<TLink> _links;
16         protected readonly TLink _sequenceLink;
17         protected readonly TLink _symbol;
18         protected TLink _total;
19
20         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink
21             ↪ sequenceLink, TLink symbol)
22         {
23             _links = links;
24             _sequenceLink = sequenceLink;
25             _symbol = symbol;
26             _total = default;
27         }
28     }
29 }

```

```

26         public virtual TLink Count()
27         {
28             if (_comparer.Compare(_total, default) > 0)
29             {
30                 return _total;
31             }
32             StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource,
33                 ↪ _links.GetTarget, IsElement, VisitElement);
34             return _total;
35         }
36
37         private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
38             ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher
39             ↪ instead of IsPartialPoint
40
41         private bool VisitElement(TLink element)
42         {
43             if (_equalityComparer.Equals(element, _symbol))
44             {
45                 _total = ArithmeticHelpers.Increment(_total);
46             }
47             return true;
48         }
49     }
50 }

```

```

./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs
1  \texttt{
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
5  {
6      public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink>,
7          ↪ TLink>
8      {
9          private readonly ILinks<TLink> _links;
10         private readonly ICriteriaMatcher<TLink> _markedSequenceMatcher;
11
12         public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
13             ↪ ICriteriaMatcher<TLink> markedSequenceMatcher)
14         {
15             _links = links;
16             _markedSequenceMatcher = markedSequenceMatcher;
17         }
18
19         public TLink Count(TLink argument) => new
20             ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
21                 ↪ _markedSequenceMatcher, argument).Count();
22     }
23 }
24 }

```

```

./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.
1  \texttt{
2  using Platform.Interfaces;
3  using Platform.Numbers;
4
5  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6  {

```

```

7 public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
  ↳ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
8 {
9     private readonly ICriteriaMatcher<TLink> _markedSequenceMatcher;
10
11     public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
12     ↳ ICriteriaMatcher<TLink> markedSequenceMatcher, TLink symbol) : base(links,
13     ↳ symbol)
14     => _markedSequenceMatcher = markedSequenceMatcher;
15
16     protected override void CountSequenceSymbolFrequency(TLink link)
17     {
18         var symbolFrequencyCounter = new
19         ↳ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
20         ↳ _markedSequenceMatcher, link, _symbol);
21         _total = ArithmeticHelpers.Add(_total, symbolFrequencyCounter.Count());
22     }
23 }
24 }
25 }

```

./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs

```

1  \texttt{
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
5  {
6      public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
7      {
8          private readonly ILinks<TLink> _links;
9          public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links =
10         ↳ links;
11          public TLink Count(TLink symbol) => new
12         ↳ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
13         ↳ symbol).Count();
14      }
15  }
16 }

```

./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs

```

1  \texttt{
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Numbers;
5
6  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7  {
8      public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11         ↳ EqualityComparer<TLink>.Default;
12         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13
14         protected readonly ILinks<TLink> _links;
15         protected readonly TLink _symbol;
16         protected readonly HashSet<TLink> _visits;
17         protected TLink _total;
18
19         public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink
20         ↳ symbol)
21         {
22             _links = links;

```

```

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

```

./Sequences/HeightProviders/CachedSequenceHeightProvider.cs

```

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

```

```

11 private readonly TLink _heightPropertyMarker;
12 private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
13 private readonly IConverter<TLink> _addressToUnaryNumberConverter;
14 private readonly IConverter<TLink> _unaryNumberToAddressConverter;
15 private readonly IPropertyOperator<TLink, TLink, TLink> _propertyOperator;
16
17 public CachedSequenceHeightProvider(
18     ILinks<TLink> links,
19     ISequenceHeightProvider<TLink> baseHeightProvider,
20     IConverter<TLink> addressToUnaryNumberConverter,
21     IConverter<TLink> unaryNumberToAddressConverter,
22     TLink heightPropertyMarker,
23     IPropertyOperator<TLink, TLink, TLink> propertyOperator)
24     : base(links)
25 {
26     _heightPropertyMarker = heightPropertyMarker;
27     _baseHeightProvider = baseHeightProvider;
28     _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
29     _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
30     _propertyOperator = propertyOperator;
31 }
32
33 public TLink Get(TLink sequence)
34 {
35     TLink height;
36     var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
37     if (_equalityComparer.Equals(heightValue, default))
38     {
39         height = _baseHeightProvider.Get(sequence);
40         heightValue = _addressToUnaryNumberConverter.Convert(height);
41         _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
42     }
43     else
44     {
45         height = _unaryNumberToAddressConverter.Convert(heightValue);
46     }
47     return height;
48 }
49 }
50 }
51 }

```

./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```

1  \texttt{
2  using Platform.Interfaces;
3  using Platform.Numbers;
4
5  namespace Platform.Data.Doublets.Sequences.HeightProviders
6  {
7      public class DefaultSequenceRightHeightProvider<TLink> :
8          ↳ LinksOperatorBase<TLink>, ISequenceHeightProvider<TLink>
9      {
10         private readonly ICreteriaMatcher<TLink> _elementMatcher;
11
12         public DefaultSequenceRightHeightProvider(ILinks<TLink> links,
13             ↳ ICreteriaMatcher<TLink> elementMatcher) : base(links) => _elementMatcher
14             ↳ = elementMatcher;
15
16         public TLink Get(TLink sequence)
17         {
18             var height = default(TLink);
19             var pairOrElement = sequence;
20             while (!_elementMatcher.IsMatched(pairOrElement))

```

```

18         {
19             pairOrElement = Links.GetTarget(pairOrElement);
20             height = ArithmeticHelpers.Increment(height);
21         }
22         return height;
23     }
24 }
25 }
26 }

```

./Sequences/HeightProviders/ISequenceHeightProvider.cs

```

1  \texttt{
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Sequences.HeightProviders
5  {
6      public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
7      {
8      }
9  }
10 }

```

./Sequences/Sequences.cs

```

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

```

```

35  /// так же проблема может быть решена при реализации дистанционных триггеров.
36  /// Нужны ли уникальные указатели вообще?
37  /// Что если обращение к информации будет происходить через содержимое всегда?
38  ///
39  /// Писать тесты.
40  ///
41  ///
42  /// Можно убрать зависимость от конкретной реализации Links,
43  /// на зависимость от абстрактного элемента, который может быть представлен
44  ↪ несколькими способами.
45  ///
46  /// Можно ли как-то сделать один общий интерфейс
47  ///
48  /// Блокчейн и/или гит для распределённой записи транзакций.
49  ///
50  </remarks>
51  public partial class Sequences : ISequences<ulong> // IList<string>, IList<ulong>]
52  ↪ (после завершения реализации Sequences)
53  {
54      private static readonly LinksCombinedConstants<bool, ulong, long> _constants =
55      ↪ Default<LinksCombinedConstants<bool, ulong, long>>.Instance;
56
57      /// <summary>Возвращает значение ulong, обозначающее любое количество
58      ↪ связей.</summary>
59      public const ulong ZeroOrMany = ulong.MaxValue;
60
61      public SequencesOptions<ulong> Options;
62      public readonly SynchronizedLinks<ulong> Links;
63      public readonly ISynchronization Sync;
64
65      public Sequences(SynchronizedLinks<ulong> links)
66      : this(links, new SequencesOptions<ulong>())
67      {
68      }
69
70      public Sequences(SynchronizedLinks<ulong> links, SequencesOptions<ulong> options)
71      {
72          Links = links;
73          Sync = links.SyncRoot;
74          Options = options;
75
76          Options.ValidateOptions();
77          Options.InitOptions(Links);
78      }
79
80      public bool IsSequence(ulong sequence)
81      {
82          return Sync.ExecuteReadOperation(() =>
83          {
84              if (Options.UseSequenceMarker)
85              {
86                  return Options.MarkedSequenceMatcher.IsMatched(sequence);
87              }
88              return !Links.Unsync.IsPartialPoint(sequence);
89          });
90      }
91
92      [MethodImpl(MethodImplOptions.AggressiveInlining)]
93      private ulong GetSequenceByElements(ulong sequence)
94      {
95          if (Options.UseSequenceMarker)
96          {
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153

```

```

94      return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
95  }
96  return sequence;
97  }
98
99  private ulong GetSequenceElements(ulong sequence)
100  {
101      if (Options.UseSequenceMarker)
102      {
103          var linkContents = new UInt64Link(Links.GetLink(sequence));
104          if (linkContents.Source == Options.SequenceMarkerLink)
105          {
106              return linkContents.Target;
107          }
108          if (linkContents.Target == Options.SequenceMarkerLink)
109          {
110              return linkContents.Source;
111          }
112      }
113      return sequence;
114  }
115
116  #region Count
117
118  public ulong Count(params ulong[] sequence)
119  {
120      if (sequence.Length == 0)
121      {
122          return Links.Count(_constants.Any, Options.SequenceMarkerLink,
123          ↪ _constants.Any);
124      }
125      if (sequence.Length == 1) // Первая связь это адрес
126      {
127          if (sequence[0] == _constants.Null)
128          {
129              return 0;
130          }
131          if (sequence[0] == _constants.Any)
132          {
133              return Count();
134          }
135          if (Options.UseSequenceMarker)
136          {
137              return Links.Count(_constants.Any, Options.SequenceMarkerLink,
138              ↪ sequence[0]);
139          }
140          return Links.Exists(sequence[0]) ? 1UL : 0;
141      }
142      throw new NotImplementedException();
143  }
144
145  private ulong CountReferences(params ulong[] restrictions)
146  {
147      if (restrictions.Length == 0)
148      {
149          return 0;
150      }
151      if (restrictions.Length == 1) // Первая связь это адрес
152      {
153          if (restrictions[0] == _constants.Null)
154          {
155              return 0;
156          }
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

154     }
155     if (Options.UseSequenceMarker)
156     {
157         var elementsLink = GetSequenceElements(restrictions[0]);
158         var sequenceLink = GetSequenceByElements(elementsLink);
159         if (sequenceLink != _constants.Null)
160         {
161             return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
162         }
163         return Links.Count(elementsLink);
164     }
165     return Links.Count(restrictions[0]);
166 }
167 throw new NotImplementedException();
168 }
169 #endregion
170 #region Create
171
172 public ulong Create(params ulong[] sequence)
173 {
174     return Sync.ExecuteWriteOperation(() =>
175     {
176         if (sequence.IsNullOrEmpty())
177         {
178             return _constants.Null;
179         }
180         Links.EnsureEachLinkExists(sequence);
181         return CreateCore(sequence);
182     });
183 }
184
185 private ulong CreateCore(params ulong[] sequence)
186 {
187     if (Options.UseIndex)
188     {
189         Options.Indexer.Index(sequence);
190     }
191     var sequenceRoot = default(ulong);
192     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
193     {
194         var matches = Each(sequence);
195         if (matches.Count > 0)
196         {
197             sequenceRoot = matches[0];
198         }
199     }
200     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
201     {
202         return CompactCore(sequence);
203     }
204     if (sequenceRoot == default)
205     {
206         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
207     }
208     if (Options.UseSequenceMarker)
209     {
210         Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
211     }
212     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами
213     ↪ элементы)

```

```

215 }
216 #endregion
217 #region Each
218
219 public List<ulong> Each(params ulong[] sequence)
220 {
221     var results = new List<ulong>();
222     Each(results.AddAndReturnTrue, sequence);
223     return results;
224 }
225
226 public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
227 {
228     return Sync.ExecuteReadOperation(() =>
229     {
230         if (sequence.IsNullOrEmpty())
231         {
232             return true;
233         }
234         Links.EnsureEachLinkIsAnyOrExists(sequence);
235         if (sequence.Count == 1)
236         {
237             var link = sequence[0];
238             if (link == _constants.Any)
239             {
240                 return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
241             }
242             return handler(link);
243         }
244         if (sequence.Count == 2)
245         {
246             return Links.Unsync.Each(sequence[0], sequence[1], handler);
247         }
248         if (Options.UseIndex && !Options.Indexer.CheckIndex(sequence))
249         {
250             return false;
251         }
252         return EachCore(handler, sequence);
253     });
254 }
255
256 private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
257 {
258     var matcher = new Matcher(this, sequence, new HashSet<LinkIndex>(), handler);
259     // TODO: Find out why matcher.HandleFullMatched executed twice for the same
260     ↪ 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],
279             ↪ sequence[sequence.Count - 1]))
280         {
281             return false;
282         }
283     }
284     return true;
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 =>
304     ↪ TryStepRightUp(handler, right, rightStep));
305
306 private bool TryStepRightUp(Func<ulong, bool> handler, ulong right, ulong
307     ↪ stepFrom)
308 {
309     var upStep = stepFrom;
310     var firstSource = Links.Unsync.GetTarget(upStep);
311     while (firstSource != right && firstSource != upStep)
312     {
313         upStep = firstSource;
314         firstSource = Links.Unsync.GetSource(upStep);
315     }
316     if (firstSource == right)
317     {
318         return handler(stepFrom);
319     }
320     return true;
321 }
322
323 private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong right) =>
324     ↪ Links.Unsync.Each(_constants.Any, right, leftStep => TryStepLeftUp(handler,
325     ↪ left, leftStep));
326
327 private bool TryStepLeftUp(Func<ulong, bool> handler, ulong left, ulong stepFrom)
328 {
329     var upStep = stepFrom;
330     var firstTarget = Links.Unsync.GetSource(upStep);
331     while (firstTarget != left && firstTarget != upStep)
332     {
333         upStep = firstTarget;
334         firstTarget = Links.Unsync.GetTarget(upStep);
335     }

```

```

331     if (firstTarget == left)
332     {
333         return handler(stepFrom);
334     }
335     return true;
336 }
337
338 #endregion
339
340 #region Update
341
342 public ulong Update(ulong[] sequence, ulong[] newSequence)
343 {
344     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
345     {
346         return _constants.Null;
347     }
348     if (sequence.IsNullOrEmpty())
349     {
350         return Create(newSequence);
351     }
352     if (newSequence.IsNullOrEmpty())
353     {
354         Delete(sequence);
355         return _constants.Null;
356     }
357     return Sync.ExecuteWriteOperation(() =>
358     {
359         Links.EnsureEachLinkIsAnyOrExists(sequence);
360         Links.EnsureEachLinkExists(newSequence);
361         return UpdateCore(sequence, newSequence);
362     });
363 }
364
365 private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
366 {
367     ulong bestVariant;
368     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
369         ↪ !sequence.EqualTo(newSequence))
370     {
371         bestVariant = CompactCore(newSequence);
372     }
373     else
374     {
375         bestVariant = CreateCore(newSequence);
376     }
377     // TODO: Check all options only ones before loop execution
378     // Возможно нужно две версии Each, возвращающий фактические
379     ↪ последовательности и с маркером,
380     // или возможно даже возвращать и тот и тот вариант. С другой стороны все
381     ↪ варианты можно получить имея только фактические последовательности.
382     foreach (var variant in Each(sequence))
383     {
384         if (variant != bestVariant)
385         {
386             UpdateOneCore(variant, bestVariant);
387         }
388     }
389     return bestVariant;
390 }
391
392 private void UpdateOneCore(ulong sequence, ulong newSequence)
393 {

```

```

391 if (Options.UseGarbageCollection)
392 {
393     var sequenceElements = GetSequenceElements(sequence);
394     var sequenceElementsContents = new
395         ↳ UInt64Link(Links.GetLink(sequenceElements));
396     var sequenceLink = GetSequenceByElements(sequenceElements);
397     var newSequenceElements = GetSequenceElements(newSequence);
398     var newSequenceLink = GetSequenceByElements(newSequenceElements);
399     if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
400     {
401         if (sequenceLink != _constants.Null)
402         {
403             Links.Unsync.Merge(sequenceLink, newSequenceLink);
404         }
405         Links.Unsync.Merge(sequenceElements, newSequenceElements);
406     }
407     ClearGarbage(sequenceElementsContents.Source);
408     ClearGarbage(sequenceElementsContents.Target);
409 }
410 else
411 {
412     if (Options.UseSequenceMarker)
413     {
414         var sequenceElements = GetSequenceElements(sequence);
415         var sequenceLink = GetSequenceByElements(sequenceElements);
416         var newSequenceElements = GetSequenceElements(newSequence);
417         var newSequenceLink = GetSequenceByElements(newSequenceElements);
418         if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
419         {
420             if (sequenceLink != _constants.Null)
421             {
422                 Links.Unsync.Merge(sequenceLink, newSequenceLink);
423             }
424             Links.Unsync.Merge(sequenceElements, newSequenceElements);
425         }
426     }
427     else
428     {
429         if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
430         {
431             Links.Unsync.Merge(sequence, newSequence);
432         }
433     }
434 }
435 }
436 #endregion
437
438 #region Delete
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

```

```

452 private void DeleteOneCore(ulong link)
453 {
454     if (Options.UseGarbageCollection)
455     {
456         var sequenceElements = GetSequenceElements(link);
457         var sequenceElementsContents = new
458             ↳ UInt64Link(Links.GetLink(sequenceElements));
459         var sequenceLink = GetSequenceByElements(sequenceElements);
460         if (Options.UseCascadeDelete || CountReferences(link) == 0)
461         {
462             if (sequenceLink != _constants.Null)
463             {
464                 Links.Unsync.Delete(sequenceLink);
465             }
466             Links.Unsync.Delete(link);
467         }
468         ClearGarbage(sequenceElementsContents.Source);
469         ClearGarbage(sequenceElementsContents.Target);
470     }
471     else
472     {
473         if (Options.UseSequenceMarker)
474         {
475             var sequenceElements = GetSequenceElements(link);
476             var sequenceLink = GetSequenceByElements(sequenceElements);
477             if (Options.UseCascadeDelete || CountReferences(link) == 0)
478             {
479                 if (sequenceLink != _constants.Null)
480                 {
481                     Links.Unsync.Delete(sequenceLink);
482                 }
483                 Links.Unsync.Delete(link);
484             }
485         }
486         else
487         {
488             if (Options.UseCascadeDelete || CountReferences(link) == 0)
489             {
490                 Links.Unsync.Delete(link);
491             }
492         }
493     }
494 }
495 #endregion
496
497 #region Compactification
498
499 /// <remarks>
500 /// bestVariant можно выбирать по максимальному числу использований,
501 /// но балансированный позволяет гарантировать уникальность (если есть
502 /// ↳ возможность,
503 /// гарантировать его использование в других местах).
504 /// Получается этот метод должен игнорировать
505 /// ↳ Options.EnforceSingleSequenceVersionOnWrite
506 /// </remarks>
507 public ulong Compact(params ulong[] sequence)
508 {
509     return Sync.ExecuteWriteOperation(() =>
510     {
511         if (sequence.IsNullOrEmpty())
512

```

```

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

```

```

572     private int _filterPosition;
573
574     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
    ↪ HashSet<LinkIndex> results, Func<LinkIndex, bool> stopableHandler,
    ↪ HashSet<LinkIndex> readAsElements = null)
    : base(sequences.Links.Unsync)
575     {
576         _sequences = sequences;
577         _patternSequence = patternSequence;
578         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x
    ↪ != _constants.Any && x != ZeroOrMany));
579         _results = results;
580         _stopableHandler = stopableHandler;
581         _readAsElements = readAsElements;
582     }
583
584     protected override bool IsElement(IList<ulong> link) => base.IsElement(link) ||
    ↪ (_readAsElements != null &&
    ↪ _readAsElements.Contains(Links.GetIndex(link))) ||
    ↪ _linksInSequence.Contains(Links.GetIndex(link));
585
586     public bool FullMatch(LinkIndex sequenceToMatch)
587     {
588         _filterPosition = 0;
589         foreach (var part in Walk(sequenceToMatch))
590         {
591             if (!FullMatchCore(Links.GetIndex(part)))
592             {
593                 break;
594             }
595         }
596         return _filterPosition == _patternSequence.Count;
597     }
598
599     private bool FullMatchCore(LinkIndex element)
600     {
601         if (_filterPosition == _patternSequence.Count)
602         {
603             _filterPosition = -2; // Длиннее чем нужно
604             return false;
605         }
606         if (_patternSequence[_filterPosition] != _constants.Any
        && element != _patternSequence[_filterPosition])
607         {
608             _filterPosition = -1;
609             return false; // Начинается/Продолжается иначе
610         }
611         _filterPosition++;
612         return true;
613     }
614
615     public void AddFullMatchedToResults(ulong sequenceToMatch)
616     {
617         if (FullMatch(sequenceToMatch))
618         {
619             _results.Add(sequenceToMatch);
620         }
621     }
622
623     public bool HandleFullMatched(ulong sequenceToMatch)
624     {
625         if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
626         {
627

```



```

628     {
629         return _stopableHandler(sequenceToMatch);
630     }
631     return true;
632 }
633
634 public bool HandleFullMatchedSequence(ulong sequenceToMatch)
635 {
636     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
637     if (sequence != _constants.Null && FullMatch(sequenceToMatch) &&
        ↪ _results.Add(sequenceToMatch))
638     {
639         return _stopableHandler(sequence);
640     }
641     return true;
642 }
643
644 ///

```

```

690     {
691         _results.Add(sequenceToMatch);
692     }
693 }
694
695 public bool HandlePartialMatched(ulong sequenceToMatch)
696 {
697     if (PartialMatch(sequenceToMatch))
698     {
699         return _stopableHandler(sequenceToMatch);
700     }
701     return true;
702 }
703
704 public void AddAllPartialMatchedToResults(IEnumerable<ulong>
    ↪ sequencesToMatch)
705 {
706     foreach (var sequenceToMatch in sequencesToMatch)
707     {
708         if (PartialMatch(sequenceToMatch))
709         {
710             _results.Add(sequenceToMatch);
711         }
712     }
713 }
714
715 public void
    ↪ AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<ulong>
    ↪ 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
728 #endregion
729 }
730 }
731 }

```

./Sequences/Sequences.Experiments.cs

```

1  \texttt{
2  using System;
3  using LinkIndex = System.UInt64;
4  using System.Collections.Generic;
5  using Stack = System.Collections.Generic.Stack<ulong>;
6  using System.Linq;
7  using System.Text;
8  using Platform.Collections;
9  using Platform.Numbers;
10 using Platform.Data.Exceptions;
11 using Platform.Data.Sequences;
12 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
13 using Platform.Data.Doublets.Sequences.Walkers;
14
15 namespace Platform.Data.Doublets.Sequences

```

```

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

```

```

75     }
76 }
77 return variants;
78 }
79
80 public List<ulong> CreateAllVariants1(params ulong[] sequence)
81 {
82     return Sync.ExecuteWriteOperation(() =>
83     {
84         if (sequence.IsNullOrEmpty())
85         {
86             return new List<ulong>();
87         }
88         Links.Unsync.EnsureEachLinkExists(sequence);
89         if (sequence.Length == 1)
90         {
91             return new List<ulong> { sequence[0] };
92         }
93         var results = new List<ulong>((int)MathHelpers.Catalan(sequence.Length));
94         return CreateAllVariants1Core(sequence, results);
95     });
96 }
97
98 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
99 {
100     if (sequence.Length == 2)
101     {
102         var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
103         if (link == _constants.Null)
104         {
105             throw new NotImplementedException("Creation cancellation is not
106             ↪ implemented.");
107         }
108         results.Add(link);
109         return results;
110     }
111     var innerSequenceLength = sequence.Length - 1;
112     var innerSequence = new ulong[innerSequenceLength];
113     for (var li = 0; li < innerSequenceLength; li++)
114     {
115         var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
116         if (link == _constants.Null)
117         {
118             throw new NotImplementedException("Creation cancellation is not
119             ↪ implemented.");
120         }
121         for (var isi = 0; isi < li; isi++)
122         {
123             innerSequence[isi] = sequence[isi];
124         }
125         innerSequence[li] = link;
126         for (var isi = li + 1; isi < innerSequenceLength; isi++)
127         {
128             innerSequence[isi] = sequence[isi + 1];
129         }
130         CreateAllVariants1Core(innerSequence, results);
131     }
132     return results;
133 }
134 #endregion

```

```

135 public HashSet<ulong> Each1(params ulong[] sequence)
136 {
137     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
138     Each1(link =>
139     {
140         if (!visitedLinks.Contains(link))
141         {
142             visitedLinks.Add(link); // изучить почему случаются повторы
143         }
144         return true;
145     }, sequence);
146     return visitedLinks;
147 }
148
149 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
150 {
151     if (sequence.Length == 2)
152     {
153         Links.Unsync.Each(sequence[0], sequence[1], handler);
154     }
155     else
156     {
157         var innerSequenceLength = sequence.Length - 1;
158         for (var li = 0; li < innerSequenceLength; li++)
159         {
160             var left = sequence[li];
161             var right = sequence[li + 1];
162             if (left == 0 && right == 0)
163             {
164                 continue;
165             }
166             var linkIndex = li;
167             ulong[] innerSequence = null;
168             Links.Unsync.Each(left, right, doublet =>
169             {
170                 if (innerSequence == null)
171                 {
172                     innerSequence = new ulong[innerSequenceLength];
173                     for (var isi = 0; isi < linkIndex; isi++)
174                     {
175                         innerSequence[isi] = sequence[isi];
176                     }
177                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
178                     {
179                         innerSequence[isi] = sequence[isi + 1];
180                     }
181                 }
182                 innerSequence[linkIndex] = doublet;
183                 Each1(handler, innerSequence);
184                 return _constants.Continue;
185             });
186         }
187     }
188 }
189
190 public HashSet<ulong> EachPart(params ulong[] sequence)
191 {
192     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
193     EachPartCore(link =>
194     {
195         if (!visitedLinks.Contains(link))
196         {

```

```

197         visitedLinks.Add(link); // изучить почему случаются повторы
198     }
199     return true;
200 }, sequence);
201 return visitedLinks;
202 }
203
204 public void EachPart(Func<ulong, bool> handler, params ulong[] sequence)
205 {
206     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
207     EachPartCore(link =>
208     {
209         if (!visitedLinks.Contains(link))
210         {
211             visitedLinks.Add(link); // изучить почему случаются повторы
212             return handler(link);
213         }
214         return true;
215     }, sequence);
216 }
217
218 private void EachPartCore(Func<ulong, bool> handler, params ulong[] sequence)
219 {
220     if (sequence.IsNullOrEmpty())
221     {
222         return;
223     }
224     Links.EnsureEachLinkIsAnyOrExists(sequence);
225     if (sequence.Length == 1)
226     {
227         var link = sequence[0];
228         if (link > 0)
229         {
230             handler(link);
231         }
232         else
233         {
234             Links.Each(_constants.Any, _constants.Any, handler);
235         }
236     }
237     else if (sequence.Length == 2)
238     {
239         // _links.Each(sequence[0], sequence[1], handler);
240         // _o | x o ...
241         // x _ | _ _ _ |
242         Links.Each(sequence[1], _constants.Any, doublet =>
243         {
244             var match = Links.SearchOrDefault(sequence[0], doublet);
245             if (match != _constants.Null)
246             {
247                 handler(match);
248             }
249             return true;
250         });
251         // | x ... x o
252         // | _ o | _ _ _ |
253         Links.Each(_constants.Any, sequence[0], doublet =>
254         {
255             var match = Links.SearchOrDefault(doublet, sequence[1]);
256             if (match != 0)
257             {
258                 handler(match);

```

```

259         }
260         return true;
261     });
262     //
263     //      . x o .
264     PartialStepRight(x => handler(x), sequence[0], sequence[1]);
265     }
266     else
267     {
268         // TODO: Implement other variants
269         return;
270     }
271 }
272
273 private void PartialStepRight(Action<ulong> handler, ulong left, ulong right)
274 {
275     Links.Unsync.Each(_constants.Any, left, doublet =>
276     {
277         StepRight(handler, doublet, right);
278         if (left != doublet)
279         {
280             PartialStepRight(handler, doublet, right);
281         }
282         return true;
283     });
284 }
285
286 private void StepRight(Action<ulong> handler, ulong left, ulong right)
287 {
288     Links.Unsync.Each(left, _constants.Any, rightStep =>
289     {
290         TryStepRightUp(handler, right, rightStep);
291         return true;
292     });
293 }
294
295 private void TryStepRightUp(Action<ulong> handler, ulong right, ulong stepFrom)
296 {
297     var upStep = stepFrom;
298     var firstSource = Links.Unsync.GetTarget(upStep);
299     while (firstSource != right && firstSource != upStep)
300     {
301         upStep = firstSource;
302         firstSource = Links.Unsync.GetSource(upStep);
303     }
304     if (firstSource == right)
305     {
306         handler(stepFrom);
307     }
308 }
309
310 // TODO: Test
311 private void PartialStepLeft(Action<ulong> handler, ulong left, ulong right)
312 {
313     Links.Unsync.Each(right, _constants.Any, doublet =>
314     {
315         StepLeft(handler, left, doublet);
316         if (right != doublet)
317         {
318             PartialStepLeft(handler, left, doublet);
319         }
320         return true;
321     });

```

```

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

```

```

385     }
386     if (sequence.Length == 2)
387     {
388         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
389         if (doublet != _constants.Null)
390         {
391             results.Add(doublet);
392         }
393         return results;
394     }
395     var linksInSequence = new HashSet<ulong>(sequence);
396     void handler(ulong result)
397     {
398         var filterPosition = 0;
399         StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
400             ↪ Links.Unsync.GetTarget,
401             ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
402             ↪ x =>
403             {
404                 if (filterPosition == sequence.Length)
405                 {
406                     filterPosition = -2; // Длиннее чем нужно
407                     return false;
408                 }
409                 if (x != sequence[filterPosition])
410                 {
411                     filterPosition = -1;
412                     return false; // Начинается иначе
413                 }
414                 filterPosition++;
415             }
416             return true;
417         });
418         if (filterPosition == sequence.Length)
419         {
420             results.Add(result);
421         }
422     }
423     if (sequence.Length >= 2)
424     {
425         StepRight(handler, sequence[0], sequence[1]);
426     }
427     var last = sequence.Length - 2;
428     for (var i = 1; i < last; i++)
429     {
430         PartialStepRight(handler, sequence[i], sequence[i + 1]);
431     }
432     if (sequence.Length >= 3)
433     {
434         StepLeft(handler, sequence[sequence.Length - 2], sequence[sequence.Length
435             ↪ - 1]);
436     }
437     }
438     return results;
439 });
440 }
441
442 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
443 {
444     return Sync.ExecuteReadOperation(() =>
445     {
446         var results = new HashSet<ulong>();

```

```

444     if (sequence.Length > 0)
445     {
446         Links.EnsureEachLinkExists(sequence);
447         var firstElement = sequence[0];
448         if (sequence.Length == 1)
449         {
450             results.Add(firstElement);
451             return results;
452         }
453     }
454     if (sequence.Length == 2)
455     {
456         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
457         if (doublet != _constants.Null)
458         {
459             results.Add(doublet);
460         }
461         return results;
462     }
463     var matcher = new Matcher(this, sequence, results, null);
464     if (sequence.Length >= 2)
465     {
466         StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
467     }
468     var last = sequence.Length - 2;
469     for (var i = 1; i < last; i++)
470     {
471         PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
472             ↪ sequence[i + 1]);
473     }
474     if (sequence.Length >= 3)
475     {
476         StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length -
477             ↪ 2], sequence[sequence.Length - 1]);
478     }
479     }
480     return results;
481 });
482 }
483
484 public const int MaxSequenceFormatSize = 200;
485
486 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[]
487     ↪ knownElements) => FormatSequence(sequenceLink, (sb, x) => sb.Append(x),
488     ↪ true, knownElements);
489
490 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
491     ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[]
492     ↪ knownElements) => Links.SyncRoot.ExecuteReadOperation(() =>
493     ↪ FormatSequence(Links.Unsync, sequenceLink, elementToString, insertComma,
494     ↪ knownElements));
495
496 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
497     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
498     ↪ LinkIndex[] knownElements)
499 {
500     var linksInSequence = new HashSet<ulong>(knownElements);
501     //var entered = new HashSet<ulong>();
502     var sb = new StringBuilder();
503     sb.Append('{');
504     if (links.Exists(sequenceLink))
505     {

```

```

495     StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource,
496     ↪ links.GetTarget,
497     ↪ x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
498     ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
499     {
500         if (insertComma && sb.Length > 1)
501         {
502             sb.Append(',');
503         }
504         //if (entered.Contains(element))
505         //{
506         //    sb.Append('{');
507         //    elementToString(sb, element);
508         //    sb.Append('}');
509         //}
510         //else
511         elementToString(sb, element);
512         if (sb.Length < MaxSequenceFormatSize)
513         {
514             return true;
515         }
516         sb.Append(insertComma ? ", ..." : "...");
517         return false;
518     });
519 }
520 sb.Append('');
521 return sb.ToString();
522 }
523
524 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
525 ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) =>
526 ↪ sb.Append(x), true, knownElements);
527
528 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
529 LinkIndex> elementToString, bool insertComma, params LinkIndex[]
530 ↪ knownElements) => Links.SyncRoot.ExecuteReadOperation(() =>
531 ↪ SafeFormatSequence(Links.Unsync, sequenceLink, elementToString,
532 ↪ insertComma, knownElements));
533
534 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
535 ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
536 ↪ LinkIndex[] knownElements)
537 {
538     var linksInSequence = new HashSet<ulong>(knownElements);
539     var entered = new HashSet<ulong>();
540     var sb = new StringBuilder();
541     sb.Append('{');
542     if (links.Exists(sequenceLink))
543     {
544         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource,
545         ↪ links.GetTarget,
546         ↪ x => linksInSequence.Contains(x) || links.IsFullPoint(x),
547         ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element
548         ↪ =>
549         {
550             if (insertComma && sb.Length > 1)
551             {
552                 sb.Append(',');
553             }
554             if (entered.Contains(element))
555             {
556                 return true;
557             }
558             elementToString(sb, element);
559             if (sb.Length < MaxSequenceFormatSize)
560             {
561                 return true;
562             }
563             sb.Append(insertComma ? ", ..." : "...");
564             return false;
565         });
566     }
567     sb.Append('');
568     return sb.ToString();
569 }

```

```

543     sb.Append('');
544     elementToString(sb, element);
545     sb.Append('');
546 }
547 else
548 {
549     elementToString(sb, element);
550 }
551 if (sb.Length < MaxSequenceFormatSize)
552 {
553     return true;
554 }
555 sb.Append(insertComma ? ", ..." : "...");
556 return false;
557 });
558 }
559 sb.Append('');
560 return sb.ToString();
561 }
562
563 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
564 {
565     return Sync.ExecuteReadOperation(() =>
566     {
567         if (sequence.Length > 0)
568         {
569             Links.EnsureEachLinkExists(sequence);
570             var results = new HashSet<ulong>();
571             for (var i = 0; i < sequence.Length; i++)
572             {
573                 AllUsagesCore(sequence[i], results);
574             }
575             var filteredResults = new List<ulong>();
576             var linksInSequence = new HashSet<ulong>(sequence);
577             foreach (var result in results)
578             {
579                 var filterPosition = -1;
580                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
581                 ↪ Links.Unsync.GetTarget,
582                 ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
583                 ↪ x =>
584                 {
585                     if (filterPosition == (sequence.Length - 1))
586                     {
587                         return false;
588                     }
589                     if (filterPosition >= 0)
590                     {
591                         if (x == sequence[filterPosition + 1])
592                         {
593                             filterPosition++;
594                         }
595                     }
596                     else
597                     {
598                         return false;
599                     }
600                 }
601             }
602             if (filterPosition < 0)
603             {
604                 if (x == sequence[0])
605                 {
606                     filterPosition = 0;
607                 }
608             }
609         }
610     });
611 }

```

```

603         }
604     }
605     return true;
606 });
607 if (filterPosition == (sequence.Length - 1))
608 {
609     filteredResults.Add(result);
610 }
611 }
612 return filteredResults;
613 }
614 return new List<ulong>();
615 });
616 }
617
618 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
619 {
620     return Sync.ExecuteReadOperation(() =>
621     {
622         if (sequence.Length > 0)
623         {
624             Links.EnsureEachLinkExists(sequence);
625             var results = new HashSet<ulong>();
626             for (var i = 0; i < sequence.Length; i++)
627             {
628                 AllUsagesCore(sequence[i], results);
629             }
630             var filteredResults = new HashSet<ulong>();
631             var matcher = new Matcher(this, sequence, filteredResults, null);
632             matcher.AddAllPartialMatchedToResults(results);
633             return filteredResults;
634         }
635         return new HashSet<ulong>();
636     });
637 }
638
639 public bool GetAllPartiallyMatchingSequences2(Func<ulong, bool> handler, params
↳ ulong[] sequence)
640 {
641     return Sync.ExecuteReadOperation(() =>
642     {
643         if (sequence.Length > 0)
644         {
645             Links.EnsureEachLinkExists(sequence);
646
647             var results = new HashSet<ulong>();
648             var filteredResults = new HashSet<ulong>();
649             var matcher = new Matcher(this, sequence, filteredResults, handler);
650             for (var i = 0; i < sequence.Length; i++)
651             {
652                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
653                 {
654                     return false;
655                 }
656             }
657             return true;
658         }
659         return true;
660     });
661 }
662
663 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[]
↳ sequence)

```

```

664 // {
665     return Sync.ExecuteReadOperation(() =>
666     {
667         if (sequence.Length > 0)
668         {
669             _links.EnsureEachLinkIsAnyOrExists(sequence);
670
671             var firstResults = new HashSet<ulong>();
672             var lastResults = new HashSet<ulong>();
673
674             var first = sequence.First(x => x != LinksConstants.Any);
675             var last = sequence.Last(x => x != LinksConstants.Any);
676
677             AllUsagesCore(first, firstResults);
678             AllUsagesCore(last, lastResults);
679
680             firstResults.IntersectWith(lastResults);
681
682             //for (var i = 0; i < sequence.Length; i++)
683             //    AllUsagesCore(sequence[i], results);
684
685             var filteredResults = new HashSet<ulong>();
686             var matcher = new Matcher(this, sequence, filteredResults, null);
687             matcher.AddAllPartialMatchedToResults(firstResults);
688             return filteredResults;
689         }
690
691         return new HashSet<ulong>();
692     });
693 // }
694
695 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
696 {
697     return Sync.ExecuteReadOperation(() =>
698     {
699         if (sequence.Length > 0)
700         {
701             Links.EnsureEachLinkIsAnyOrExists(sequence);
702             var firstResults = new HashSet<ulong>();
703             var lastResults = new HashSet<ulong>();
704             var first = sequence.First(x => x != _constants.Any);
705             var last = sequence.Last(x => x != _constants.Any);
706             AllUsagesCore(first, firstResults);
707             AllUsagesCore(last, lastResults);
708             firstResults.IntersectWith(lastResults);
709             //for (var i = 0; i < sequence.Length; i++)
710             //    AllUsagesCore(sequence[i], results);
711             var filteredResults = new HashSet<ulong>();
712             var matcher = new Matcher(this, sequence, filteredResults, null);
713             matcher.AddAllPartialMatchedToResults(firstResults);
714             return filteredResults;
715         }
716         return new HashSet<ulong>();
717     });
718 }
719
720 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong>
↳ readAsElements, IList<ulong> sequence)
721 {
722     return Sync.ExecuteReadOperation(() =>
723     {
724         if (sequence.Count > 0)

```

```

725 {
726     Links.EnsureEachLinkExists(sequence);
727     var results = new HashSet<LinkIndex>();
728     //var nextResults = new HashSet<ulong>();
729     for (var i = 0; i < sequence.Length; i++)
730     {
731         AllUsagesCore(sequence[i], nextResults);
732         if (results.IsNullOrEmpty())
733         {
734             results = nextResults;
735             nextResults = new HashSet<ulong>();
736         }
737         else
738         {
739             results.IntersectWith(nextResults);
740             nextResults.Clear();
741         }
742     }
743     var collector1 = new AllUsagesCollector1(Links.Unsync, results);
744     collector1.Collect(Links.Unsync.GetLink(sequence[0]));
745     var next = new HashSet<ulong>();
746     for (var i = 1; i < sequence.Count; i++)
747     {
748         var collector = new AllUsagesCollector1(Links.Unsync, next);
749         collector.Collect(Links.Unsync.GetLink(sequence[i]));
750
751         results.IntersectWith(next);
752         next.Clear();
753     }
754     var filteredResults = new HashSet<ulong>();
755     var matcher = new Matcher(this, sequence, filteredResults, null,
756         ↪ readAsElements);
757     matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x
758         ↪ => x)); // OrderBy is a Hack
759     return filteredResults;
760 }
761
762 // Does not work
763 public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong>
764     ↪ readAsElements, params ulong[] sequence)
765 {
766     var visited = new HashSet<ulong>();
767     var results = new HashSet<ulong>();
768     var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
769         ↪ true; }, readAsElements);
770     var last = sequence.Length - 1;
771     for (var i = 0; i < last; i++)
772     {
773         PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
774     }
775     return results;
776 }
777
778 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
779 {
780     return Sync.ExecuteReadOperation(() =>
781     {
782         if (sequence.Length > 0)

```

```

783 Links.EnsureEachLinkExists(sequence);
784 //var firstElement = sequence[0];
785 //if (sequence.Length == 1)
786 // {
787 //     //results.Add(firstElement);
788 //     return results;
789 // }
790 //if (sequence.Length == 2)
791 // {
792 //     //var doublet = links.SearchCore(firstElement, sequence[1]);
793 //     //if (doublet != Doublets.Links.Null)
794 //         results.Add(doublet);
795 //     return results;
796 // }
797 //var lastElement = sequence[sequence.Length - 1];
798 //Func<ulong, bool> handler = x =>
799 // {
800 //     if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
801 //         ↪ results.Add(x);
802 //     return true;
803 // };
804 //if (sequence.Length >= 2)
805 //     StepRight(handler, sequence[0], sequence[1]);
806 //var last = sequence.Length - 2;
807 //for (var i = 1; i < last; i++)
808 //     PartialStepRight(handler, sequence[i], sequence[i + 1]);
809 //if (sequence.Length >= 3)
810 //     StepLeft(handler, sequence[sequence.Length - 2],
811 //         ↪ sequence[sequence.Length - 1]);
812 //if (sequence.Length == 1)
813 // {
814 //     throw new NotImplementedException(); // all sequences, containing
815 //     ↪ this element?
816 // }
817 //if (sequence.Length == 2)
818 // {
819 //     var results = new List<ulong>();
820 //     PartialStepRight(results.Add, sequence[0], sequence[1]);
821 //     return results;
822 // }
823 //var matches = new List<List<ulong>>();
824 //var last = sequence.Length - 1;
825 //for (var i = 0; i < last; i++)
826 // {
827 //     var results = new List<ulong>();
828 //     //StepRight(results.Add, sequence[i], sequence[i + 1]);
829 //     PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
830 //     if (results.Count > 0)
831 //         matches.Add(results);
832 //     else
833 //         return results;
834 //     if (matches.Count == 2)
835 //     {
836 //         var merged = new List<ulong>();
837 //         for (var j = 0; j < matches[0].Count; j++)
838 //             for (var k = 0; k < matches[1].Count; k++)
839 //                 CloseInnerConnections(merged.Add, matches[0][j],
840 //                     ↪ matches[1][k]);
841 //         if (merged.Count > 0)
842 //             matches = new List<List<ulong>> { merged };

```



```

839         ////////////// else
840         return new List<ulong>();
841     }
842     ////////////// }
843     if (matches.Count > 0)
844     {
845         var usages = new HashSet<ulong>();
846         for (int i = 0; i < sequence.Length; i++)
847         {
848             AllUsagesCore(sequence[i], usages);
849         }
850         //for (int i = 0; i < matches[0].Count; i++)
851         //    AllUsagesCore(matches[0][i], usages);
852         //usages.UnionWith(matches[0]);
853         return usages.ToList();
854     }
855     var firstLinkUsages = new HashSet<ulong>();
856     AllUsagesCore(sequence[0], firstLinkUsages);
857     firstLinkUsages.Add(sequence[0]);
858     //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
859     //    sequence[0] }; // or all sequences, containing this element?
860     //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
861     //    1).ToList();
862     var results = new HashSet<ulong>();
863     foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
864         firstLinkUsages, 1))
865     {
866         AllUsagesCore(match, results);
867     }
868     return results.ToList();
869 }
870
871 /// <remarks>
872 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
873 /// </remarks>
874 public HashSet<ulong> AllUsages(ulong link)
875 {
876     return Sync.ExecuteReadOperation(() =>
877     {
878         var usages = new HashSet<ulong>();
879         AllUsagesCore(link, usages);
880         return usages;
881     });
882 }
883
884 // При сборе всех использований (последовательностей) можно сохранять
885 // обратный путь к той связи с которой начинался поиск (STTTSSSTT),
886 // причём достаточно одного бита для хранения перехода влево или вправо
887 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
888 {
889     bool handler(ulong doublet)
890     {
891         if (usages.Add(doublet))
892         {
893             AllUsagesCore(doublet, usages);
894         }
895         return true;
896     }
897     Links.Unsync.Each(link, _constants.Any, handler);

```

```

897     Links.Unsync.Each(_constants.Any, link, handler);
898 }
899
900 public HashSet<ulong> AllBottomUsages(ulong link)
901 {
902     return Sync.ExecuteReadOperation(() =>
903     {
904         var visits = new HashSet<ulong>();
905         var usages = new HashSet<ulong>();
906         AllBottomUsagesCore(link, visits, usages);
907         return usages;
908     });
909 }
910
911 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits,
912     HashSet<ulong> usages)
913 {
914     bool handler(ulong doublet)
915     {
916         if (visits.Add(doublet))
917         {
918             AllBottomUsagesCore(doublet, visits, usages);
919         }
920         return true;
921     }
922     if (Links.Unsync.Count(_constants.Any, link) == 0)
923     {
924         usages.Add(link);
925     }
926     else
927     {
928         Links.Unsync.Each(link, _constants.Any, handler);
929         Links.Unsync.Each(_constants.Any, link, handler);
930     }
931 }
932
933 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
934 {
935     if (Options.UseSequenceMarker)
936     {
937         var counter = new
938             TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
939             Options.MarkedSequenceMatcher, symbol);
940         return counter.Count();
941     }
942     else
943     {
944         var counter = new
945             TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links, symbol);
946         return counter.Count();
947     }
948 }
949
950 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool>
951     outerHandler)
952 {
953     bool handler(ulong doublet)
954     {
955         if (usages.Add(doublet))
956         {
957             if (!outerHandler(doublet))
958             {

```

```

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,
991         ↪ CalculateCore);
992
993     private bool CalculateCore(ulong link)
994     {
995         if (_totals[link] == 0)
996         {
997             var total = 1UL;
998             _totals[link] = total;
999             var visitedChildren = new HashSet<ulong>();
1000             bool linkCalculator(ulong child)
1001             {
1002                 if (link != child && visitedChildren.Add(child))
1003                 {
1004                     total += _totals[child] == 0 ? 1 : _totals[child];
1005                 }
1006                 return true;
1007             }
1008             _links.Unsync.Each(link, _constants.Any, linkCalculator);
1009             _links.Unsync.Each(_constants.Any, link, linkCalculator);
1010             _totals[link] = total;
1011         }
1012         return true;
1013     }
1014 }
1015
1016 private class AllUsagesCalculator2

```

```

1016 {
1017     private readonly SynchronizedLinks<ulong> _links;
1018     private readonly ulong[] _totals;
1019
1020     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1021     {
1022         _links = links;
1023         _totals = totals;
1024     }
1025
1026     public void Calculate() => _links.Each(_constants.Any, _constants.Any,
1027         ↪ CalculateCore);
1028
1029     private bool IsElement(ulong link)
1030     {
1031         // _linksInSequence.Contains(link) ||
1032         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link)
1033             ↪ == link;
1034     }
1035
1036     private bool CalculateCore(ulong link)
1037     {
1038         // TODO: Проработать защиту от заикливания
1039         // Основано на SequenceWalker.WalkLeft
1040         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1041         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1042         Func<ulong, bool> isElement = IsElement;
1043         void visitLeaf(ulong parent)
1044         {
1045             if (link != parent)
1046             {
1047                 _totals[parent]++;
1048             }
1049         }
1050         void visitNode(ulong parent)
1051         {
1052             if (link != parent)
1053             {
1054                 _totals[parent]++;
1055             }
1056         }
1057         var stack = new Stack();
1058         var element = link;
1059         if (isElement(element))
1060         {
1061             visitLeaf(element);
1062         }
1063         else
1064         {
1065             while (true)
1066             {
1067                 if (isElement(element))
1068                 {
1069                     if (stack.Count == 0)
1070                     {
1071                         break;
1072                     }
1073                     element = stack.Pop();
1074                     var source = getSource(element);
1075                     var target = getTarget(element);
1076                     // Обработка элемента
1077                     if (isElement(target))

```

```

1076         {
1077             visitLeaf(target);
1078         }
1079         if (isElement(source))
1080         {
1081             visitLeaf(source);
1082         }
1083         element = source;
1084     }
1085     else
1086     {
1087         stack.Push(element);
1088         visitNode(element);
1089         element = getTarget(element);
1090     }
1091 }
1092 }
1093 totals[link]++;
1094 return true;
1095 }
1096 }
1097
1098 private class AllUsagesCollector
1099 {
1100     private readonly ILinks<ulong> _links;
1101     private readonly HashSet<ulong> _usages;
1102
1103     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1104     {
1105         _links = links;
1106         _usages = usages;
1107     }
1108
1109     public bool Collect(ulong link)
1110     {
1111         if (_usages.Add(link))
1112         {
1113             _links.Each(link, _constants.Any, Collect);
1114             _links.Each(_constants.Any, link, Collect);
1115         }
1116         return true;
1117     }
1118 }
1119
1120 private class AllUsagesCollector1
1121 {
1122     private readonly ILinks<ulong> _links;
1123     private readonly HashSet<ulong> _usages;
1124     private readonly ulong _continue;
1125
1126     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1127     {
1128         _links = links;
1129         _usages = usages;
1130         _continue = _links.Constants.Continue;
1131     }
1132
1133     public ulong Collect(ICollection<ulong> link)
1134     {
1135         var linkIndex = _links.GetIndex(link);
1136         if (_usages.Add(linkIndex))
1137         {
1138             _links.Each(Collect, _constants.Any, linkIndex);
1139         }

```

```

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

```

```

1202 private void AllCloseConnections(Action<ulong> handler, ulong left, ulong right)
1203 {
1204     // Direct
1205     if (left == right)
1206     {
1207         handler(left);
1208     }
1209     var doublet = Links.Unsync.SearchOrDefault(left, right);
1210     if (doublet != _constants.Null)
1211     {
1212         handler(doublet);
1213     }
1214     // Inner
1215     CloseInnerConnections(handler, left, right);
1216     // Outer
1217     StepLeft(handler, left, right);
1218     StepRight(handler, left, right);
1219     PartialStepRight(handler, left, right);
1220     PartialStepLeft(handler, left, right);
1221 }
1222
1223 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
    ↳ HashSet<ulong> previousMatchings, long startAt)
1224 {
1225     if (startAt >= sequence.Length) // ?
1226     {
1227         return previousMatchings;
1228     }
1229     var secondLinkUsages = new HashSet<ulong>();
1230     AllUsagesCore(sequence[startAt], secondLinkUsages);
1231     secondLinkUsages.Add(sequence[startAt]);
1232     var matchings = new HashSet<ulong>();
1233     //for (var i = 0; i < previousMatchings.Count; i++)
1234     foreach (var secondLinkUsage in secondLinkUsages)
1235     {
1236         foreach (var previousMatching in previousMatchings)
1237         {
1238             // AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
    ↳ secondLinkUsage);
1239             StepRight(matchings.AddAndReturnVoid, previousMatching,
    ↳ secondLinkUsage);
1240             TryStepRightUp(matchings.AddAndReturnVoid, secondLinkUsage,
    ↳ previousMatching);
1241             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
    ↳ sequence[startAt]); // почему-то эта ошибочная запись приводит к
    ↳ желаемым результатам.
1242             PartialStepRight(matchings.AddAndReturnVoid, previousMatching,
    ↳ secondLinkUsage);
1243         }
1244     }
1245     if (matchings.Count == 0)
1246     {
1247         return matchings;
1248     }
1249     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1);
    ↳ // ??
1250 }
1251
1252 private static void
    ↳ EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong> links,
    ↳ params ulong[] sequence)

```

```

1253 {
1254     if (sequence == null)
1255     {
1256         return;
1257     }
1258     for (var i = 0; i < sequence.Length; i++)
1259     {
1260         if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany &&
    ↳ !links.Exists(sequence[i]))
1261         {
1262             throw new ArgumentLinkDoesNotExistException<ulong>(sequence[i],
    ↳ $"patternSequence[{i}]");
1263         }
1264     }
1265 }
1266
1267 // Pattern Matching -> Key To Triggers
1268 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1269 {
1270     return Sync.ExecuteReadOperation(() =>
1271     {
1272         patternSequence = Simplify(patternSequence);
1273         if (patternSequence.Length > 0)
1274         {
1275             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1276             var uniqueSequenceElements = new HashSet<ulong>();
1277             for (var i = 0; i < patternSequence.Length; i++)
1278             {
1279                 if (patternSequence[i] != _constants.Any && patternSequence[i] !=
    ↳ ZeroOrMany)
1280                 {
1281                     uniqueSequenceElements.Add(patternSequence[i]);
1282                 }
1283             }
1284             var results = new HashSet<ulong>();
1285             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1286             {
1287                 AllUsagesCore(uniqueSequenceElement, results);
1288             }
1289             var filteredResults = new HashSet<ulong>();
1290             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1291             matcher.AddAllPatternMatchedToResults(results);
1292             return filteredResults;
1293         }
1294         return new HashSet<ulong>();
1295     });
1296 }
1297
1298 // Найти все возможные связи между указанным списком связей.
1299 // Находит связи между всеми указанными связями в любом порядке.
1300 // TODO: решить что делать с повторами (когда одни и те же элементы
    ↳ встречаются несколько раз в последовательности)
1301 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1302 {
1303     return Sync.ExecuteReadOperation(() =>
1304     {
1305         var results = new HashSet<ulong>();
1306         if (linksToConnect.Length > 0)
1307         {
1308             Links.EnsureEachLinkExists(linksToConnect);
1309             AllUsagesCore(linksToConnect[0], results);

```

```

1310         for (var i = 1; i < linksToConnect.Length; i++)
1311         {
1312             var next = new HashSet<ulong>();
1313             AllUsagesCore(linksToConnect[i], next);
1314             results.IntersectWith(next);
1315         }
1316     }
1317     return results;
1318 });
1319 }
1320
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);
1353             collector1.Collect(linksToConnect[0]);
1354             //AllUsagesCore(linksToConnect[0], results);
1355             for (var i = 1; i < linksToConnect.Length; i++)
1356             {
1357                 var next = new HashSet<ulong>();
1358                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1359                 collector.Collect(linksToConnect[i]);
1360                 //AllUsagesCore(linksToConnect[i], next);
1361                 //results.IntersectWith(next);
1362                 results = next;
1363             }
1364             return results;
1365         }
1366     });
1367 }
1368
1369 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1370 {
1371     return Sync.ExecuteReadOperation(() =>

```

```

1372     {
1373         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1374         ↪ BitArray((int) links.Total + 1);
1375         if (linksToConnect.Length > 0)
1376         {
1377             Links.EnsureEachLinkExists(linksToConnect);
1378             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1379             collector1.Collect(linksToConnect[0]);
1380             for (var i = 1; i < linksToConnect.Length; i++)
1381             {
1382                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1383                 ↪ BitArray((int) _links.Total + 1);
1384                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1385                 collector.Collect(linksToConnect[i]);
1386                 results = results.And(next);
1387             }
1388             return results.GetSetUInt64Indices();
1389         }
1390     });
1391 }
1392
1393 private static ulong[] Simplify(ulong[] sequence)
1394 {
1395     // Считаем новый размер последовательности
1396     long newLength = 0;
1397     var zeroOrManyStepped = false;
1398     for (var i = 0; i < sequence.Length; i++)
1399     {
1400         if (sequence[i] == ZeroOrMany)
1401         {
1402             if (zeroOrManyStepped)
1403             {
1404                 continue;
1405             }
1406             zeroOrManyStepped = true;
1407         }
1408         else
1409         {
1410             //if (zeroOrManyStepped) Is it efficient?
1411             zeroOrManyStepped = false;
1412             newLength++;
1413         }
1414     }
1415     // Строим новую последовательность
1416     zeroOrManyStepped = false;
1417     var newSequence = new ulong[newLength];
1418     long j = 0;
1419     for (var i = 0; i < sequence.Length; i++)
1420     {
1421         //var current = zeroOrManyStepped;
1422         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1423         //if (current && zeroOrManyStepped)
1424         //    continue;
1425         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1426         //if (zeroOrManyStepped && newZeroOrManyStepped)
1427         //    continue;
1428         //zeroOrManyStepped = newZeroOrManyStepped;
1429         if (sequence[i] == ZeroOrMany)
1430         {
1431             if (zeroOrManyStepped)
1432             {
1433                 continue;

```

```

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

```

```

1492         var element = elements[i];
1493         if (element != 0)
1494         {
1495             CollectMatchingSequences(element, leftBound + 1, middleLinks,
1496             ↪ rightLink, rightBound, ref results);
1497         }
1498     }
1499     else
1500     {
1501         for (var i = elements.Length - 1; i >= 0; i--)
1502         {
1503             var element = elements[i];
1504             if (element != 0)
1505             {
1506                 results.Add(element);
1507             }
1508         }
1509     }
1510 }
1511 else
1512 {
1513     var nextRightLink = middleLinks[rightBound];
1514     var elements = GetLeftElements(rightLink, nextRightLink);
1515     if (leftBound <= rightBound)
1516     {
1517         for (var i = elements.Length - 1; i >= 0; i--)
1518         {
1519             var element = elements[i];
1520             if (element != 0)
1521             {
1522                 CollectMatchingSequences(leftLink, leftBound, middleLinks, elements[i],
1523                 ↪ rightBound - 1, ref results);
1524             }
1525         }
1526     }
1527     else
1528     {
1529         for (var i = elements.Length - 1; i >= 0; i--)
1530         {
1531             var element = elements[i];
1532             if (element != 0)
1533             {
1534                 results.Add(element);
1535             }
1536         }
1537     }
1538 }
1539 }
1540
1541 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1542 {
1543     var result = new ulong[5];
1544     TryStepRight(startLink, rightLink, result, 0);
1545     Links.Each(_constants.Any, startLink, couple =>
1546     {
1547         if (couple != startLink)
1548         {
1549             if (TryStepRight(couple, rightLink, result, 2))
1550             {

```

```

1551     }
1552 }
1553 return true;
1554 });
1555 if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1556 {
1557     result[4] = startLink;
1558 }
1559 return result;
1560 }
1561
1562 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1563 {
1564     var added = 0;
1565     Links.Each(startLink, _constants.Any, couple =>
1566     {
1567         if (couple != startLink)
1568         {
1569             var coupleTarget = Links.GetTarget(couple);
1570             if (coupleTarget == rightLink)
1571             {
1572                 result[offset] = couple;
1573                 if (++added == 2)
1574                 {
1575                     return false;
1576                 }
1577             }
1578             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1579                 ↪ == Net.And &&
1580             {
1581                 result[offset + 1] = couple;
1582                 if (++added == 2)
1583                 {
1584                     return false;
1585                 }
1586             }
1587         }
1588     });
1589     return added > 0;
1590 }
1591
1592 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1593 {
1594     var result = new ulong[5];
1595     TryStepLeft(startLink, leftLink, result, 0);
1596     Links.Each(startLink, _constants.Any, couple =>
1597     {
1598         if (couple != startLink)
1599         {
1600             if (TryStepLeft(couple, leftLink, result, 2))
1601             {
1602                 return false;
1603             }
1604         }
1605     });
1606     return result;
1607 }
1608 if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1609 {
1610     result[4] = leftLink;
1611 }
1612 return result;
1613 }

```

```

1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674

public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
{
    var added = 0;
    Links.Each(_constants.Any, startLink, couple =>
    {
        if (couple != startLink)
        {
            var coupleSource = Links.GetSource(couple);
            if (coupleSource == leftLink)
            {
                result[offset] = couple;
                if (++added == 2)
                {
                    return false;
                }
            }
            else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1640 ↪ == Net.And &&
            {
                result[offset + 1] = couple;
                if (++added == 2)
                {
                    return false;
                }
            }
        }
    });
    return added > 0;
}

#endregion

#region Walkers

public class PatternMatcher : RightSequenceWalker<ulong>
{
    private readonly Sequences _sequences;
    private readonly ulong[] _patternSequence;
    private readonly HashSet<LinkIndex> _linksInSequence;
    private readonly HashSet<LinkIndex> _results;

    #region Pattern Match

    enum PatternBlockType
    {
        Undefined,
        Gap,
        Elements
    }

    struct PatternBlock
    {
        public PatternBlockType Type;
        public long Start;
        public long Stop;
    }

    private readonly List<PatternBlock> _pattern;
    private int _patternPosition;
    private long _sequencePosition;

```

```

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

```

```

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 if (patternBlock.Type != PatternBlockType.Undefined)
1784 {
1785     pattern.Add(patternBlock);
1786 }
1787 return pattern;
1788 }
1789
1790 /// * match: search for regexp anywhere in text */
1791 int match(char* regexp, char* text)
1792 {
1793     do
1794     {
1795         while (*text++ != '\0');

```



```

1796 // return 0;
1797 //}
1798
1799 // * matchhere: search for regexp at beginning of text */
1800 // int matchhere(char* regexp, char* text)
1801 // {
1802 //     if (regexp[0] == '\0')
1803 //         return 1;
1804 //     if (regexp[1] == '*')
1805 //         return matchstar(regexp[0], regexp + 2, text);
1806 //     if (regexp[0] == '$' && regexp[1] == '\0')
1807 //         return *text == '\0';
1808 //     if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1809 //         return matchhere(regexp + 1, text + 1);
1810 //     return 0;
1811 //}
1812
1813 // * matchstar: search for c*regexp at beginning of text */
1814 // int matchstar(int c, char* regexp, char* text)
1815 // {
1816 //     do
1817 //     {
1818 //         /* a * matches zero or more instances */
1819 //         if (matchhere(regexp, text))
1820 //             return 1;
1821 //     } while (*text != '\0' && (*text++ == c || c == '.'));
1822 //     return 0;
1823 //}
1824
1825 // private void GetNextPatternElement(out LinkIndex element, out long
1826 //     ↳ mininumGap, out long maximumGap)
1827 // {
1828 //     mininumGap = 0;
1829 //     maximumGap = 0;
1830 //     element = 0;
1831 //     for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1832 //     {
1833 //         if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1834 //             mininumGap++;
1835 //         else if (_patternSequence[_patternPosition] == ZeroOrMany)
1836 //             maximumGap = long.MaxValue;
1837 //         else
1838 //             break;
1839 //     }
1840
1841 //     if (maximumGap < mininumGap)
1842 //         maximumGap = mininumGap;
1843 //}
1844
1845 private bool PatternMatchCore(LinkIndex element)
1846 {
1847     if (_patternPosition >= _pattern.Count)
1848     {
1849         _patternPosition = -2;
1850         return false;
1851     }
1852     var currentPatternBlock = _pattern[_patternPosition];
1853     if (currentPatternBlock.Type == PatternBlockType.Gap)
1854     {
1855         // var currentMatchingBlockLength = (_sequencePosition -
1856         //     ↳ lastMatchedBlockPosition);
1857         if (_sequencePosition < currentPatternBlock.Start)
1858         {

```

```

1856         _sequencePosition++;
1857         return true; // Двигаемся дальше
1858     }
1859     // Это последний блок
1860     if (_pattern.Count == _patternPosition + 1)
1861     {
1862         _patternPosition++;
1863         _sequencePosition = 0;
1864         return false; // Полное соответствие
1865     }
1866     else
1867     {
1868         if (_sequencePosition > currentPatternBlock.Stop)
1869         {
1870             return false; // Соответствие невозможно
1871         }
1872         var nextPatternBlock = _pattern[_patternPosition + 1];
1873         if (_patternSequence[nextPatternBlock.Start] == element)
1874         {
1875             if (nextPatternBlock.Start < nextPatternBlock.Stop)
1876             {
1877                 _patternPosition++;
1878                 _sequencePosition = 1;
1879             }
1880             else
1881             {
1882                 _patternPosition += 2;
1883                 _sequencePosition = 0;
1884             }
1885         }
1886     }
1887 }
1888
1889 else // currentPatternBlock.Type == PatternBlockType.Elements
1890 {
1891     var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1892     if (_patternSequence[patternElementPosition] != element)
1893     {
1894         return false; // Соответствие невозможно
1895     }
1896     if (patternElementPosition == currentPatternBlock.Stop)
1897     {
1898         _patternPosition++;
1899         _sequencePosition = 0;
1900     }
1901     else
1902     {
1903         _sequencePosition++;
1904     }
1905 }
1906
1907 return true;
1908 // if (_patternSequence[_patternPosition] != element)
1909 //     return false;
1910 // else
1911 // {
1912 //     _sequencePosition++;
1913 //     _patternPosition++;
1914 //     return true;
1915 // }
1916 //}
1917
1918 // if (_filterPosition == _patternSequence.Length)
1919 // {
1920 //     _filterPosition = -2; // Длиннее чем нужно

```

```

1918         // return false;
1919     //}
1920     //if (element != _patternSequence[_filterPosition])
1921     //{
1922         _filterPosition = -1;
1923         // return false; // Начинается иначе
1924     //}
1925     //filterPosition++;
1926     //if (_filterPosition == (_patternSequence.Length - 1))
1927     //    return false;
1928     //if (_filterPosition >= 0)
1929     //{
1930         if (element == _patternSequence[_filterPosition + 1])
1931             _filterPosition++;
1932         else
1933             return false;
1934     //}
1935     //if (_filterPosition < 0)
1936     //{
1937         if (element == _patternSequence[0])
1938             _filterPosition = 0;
1939     //}
1940 }
1941
1942 public void AddAllPatternMatchedToResults(IEnumerable<ulong>
↵ sequencesToMatch)
1943 {
1944     foreach (var sequenceToMatch in sequencesToMatch)
1945     {
1946         if (PatternMatch(sequenceToMatch))
1947         {
1948             _results.Add(sequenceToMatch);
1949         }
1950     }
1951 }
1952 }
1953 #endregion
1954 }
1955 }
1956 }
1957 }

```

./Sequences/Sequences.Experiments.ReadSequence.cs

```

1  [texttt{
2  // #define USEARRAYPOOL
3  using System;
4  using System.Runtime.CompilerServices;
5  #if USEARRAYPOOL
6  using Platform.Collections;
7  #endif
8
9  namespace Platform.Data.Doublets.Sequences
10 {
11     partial class Sequences
12     {
13         public ulong[] ReadSequenceCore(ulong sequence, Func<ulong, bool> isElement)
14         {
15             var links = Links.Unsync;
16             var length = 1;
17             var array = new ulong[length];
18             array[0] = sequence;
19
20             if (isElement(sequence))

```

```

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

```

```

83     var finalArray = new ulong[filledElementsCount];
84     for (int i = 0, j = 0; i < array.Length; i++)
85     {
86         if (array[i] > 0)
87         {
88             finalArray[j] = array[i];
89             j++;
90         }
91     }
92     #if USEARRAYPOOL
93         ArrayPool.Free(array);
94     #endif
95     return finalArray;
96 }
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 private static int CountFilledElements(ulong[] array)
100 {
101     var count = 0;
102     for (var i = 0; i < array.Length; i++)
103     {
104         if (array[i] > 0)
105         {
106             count++;
107         }
108     }
109     return count;
110 }
111 }
112 }
113 }

```

./Sequences/SequencesExtensions.cs

```

1  [!texttt{
2  using Platform.Data.Sequences;
3  using System.Collections.Generic;
4
5  namespace Platform.Data.Doublets.Sequences
6  {
7      public static class SequencesExtensions
8      {
9          public static TLink Create<TLink>(this ISequences<TLink> sequences,
10             ↳ IList<TLink[]> groupedSequence)
11          {
12              var finalSequence = new TLink[groupedSequence.Count];
13              for (var i = 0; i < finalSequence.Length; i++)
14              {
15                  var part = groupedSequence[i];
16                  finalSequence[i] = part.Length == 1 ? part[0] : sequences.Create(part);
17              }
18              return sequences.Create(finalSequence);
19          }
20      }
21  }

```

./Sequences/SequencesIndexer.cs

```

1  [!texttt{
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets.Sequences
5  {

```

```

6  public class SequencesIndexer<TLink>
7  {
8      private static readonly EqualityComparer<TLink> _equalityComparer =
9          ↳ EqualityComparer<TLink>.Default;
10
11      private readonly ISynchronizedLinks<TLink> _links;
12      private readonly TLink _null;
13
14      public SequencesIndexer(ISynchronizedLinks<TLink> links)
15      {
16          _links = links;
17          _null = _links.Constants.Null;
18      }
19
20      /// <summary>
21      /// Индексирует последовательность глобально, и возвращает значение,
22      /// определяющие была ли запрошенная последовательность проиндексирована
23      /// ранее.
24      /// </summary>
25      /// <param name="sequence">Последовательность для индексации.</param>
26      /// <returns>
27      /// True если последовательность уже была проиндексирована ранее и
28      /// False если последовательность была проиндексирована только что.
29      /// </returns>
30      public bool Index(TLink[] sequence)
31      {
32          var indexed = true;
33          var i = sequence.Length;
34          while (--i >= 1 && (indexed =
35             ↳ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1],
36             ↳ sequence[i]), _null))) { }
37          for (; i >= 1; i--)
38          {
39              _links.GetOrCreate(sequence[i - 1], sequence[i]);
40          }
41          return indexed;
42      }
43
44      public bool BulkIndex(TLink[] sequence)
45      {
46          var indexed = true;
47          var i = sequence.Length;
48          var links = _links.Unsync;
49          links.SyncRoot.ExecuteReadOperation(() =>
50          {
51              while (--i >= 1 && (indexed =
52                 ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
53                 ↳ sequence[i]), _null))) { }
54          });
55          if (indexed == false)
56          {
57              links.SyncRoot.ExecuteWriteOperation(() =>
58              {
59                  for (; i >= 1; i--)
60                  {
61                      links.GetOrCreate(sequence[i - 1], sequence[i]);
62                  }
63              });
64          }
65          return indexed;
66      }
67
68      public bool BulkIndexUnsync(TLink[] sequence)

```

```

63     {
64         var indexed = true;
65         var i = sequence.Length;
66         var links = _links.Unsync;
67         while (--i >= 1 && (indexed =
            ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
            ↳ sequence[i]), _null))) { }
68         for (; i >= 1; i--)
69         {
70             links.GetOrCreate(sequence[i - 1], sequence[i]);
71         }
72         return indexed;
73     }
74
75     public bool CheckIndex(IList<TLink> sequence)
76     {
77         var indexed = true;
78         var i = sequence.Count;
79         while (--i >= 1 && (indexed =
            ↳ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1],
            ↳ sequence[i]), _null))) { }
80         return indexed;
81     }
82 }
83
84 }

```

./Sequences/SequencesOptions.cs

```

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

```

```

34
35 public void InitOptions(ISynchronizedLinks<TLink> links)
36 {
37     if (UseSequenceMarker)
38     {
39         if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
40         {
41             SequenceMarkerLink = links.CreatePoint();
42         }
43         else
44         {
45             if (!links.Exists(SequenceMarkerLink))
46             {
47                 var link = links.CreatePoint();
48                 if (!_equalityComparer.Equals(link, SequenceMarkerLink))
49                 {
50                     throw new InvalidOperationException("Cannot recreate sequence
                        ↳ marker link.");
51                 }
52             }
53         }
54     }
55     if (MarkedSequenceMatcher == null)
56     {
57         MarkedSequenceMatcher = new
            ↳ MarkedSequenceCriteriaMatcher<TLink>(links, SequenceMarkerLink);
58     }
59     var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
60     if (UseCompression)
61     {
62         if (LinksToSequenceConverter == null)
63         {
64             ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
65             if (UseSequenceMarker)
66             {
67                 totalSequenceSymbolFrequencyCounter = new
                    ↳ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                    ↳ MarkedSequenceMatcher);
68             }
69             else
70             {
71                 totalSequenceSymbolFrequencyCounter = new
                    ↳ TotalSequenceSymbolFrequencyCounter<TLink>(links);
72             }
73             var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                ↳ totalSequenceSymbolFrequencyCounter);
74             var compressingConverter = new CompressingConverter<TLink>(links,
                ↳ balancedVariantConverter, doubletFrequenciesCache);
75             LinksToSequenceConverter = compressingConverter;
76         }
77     }
78     else
79     {
80         if (LinksToSequenceConverter == null)
81         {
82             LinksToSequenceConverter = balancedVariantConverter;
83         }
84     }
85     if (UseIndex && Indexer == null)
86     {
87         Indexer = new SequencesIndexer<TLink>(links);

```

```

88     }
89 }
90
91 public void ValidateOptions()
92 {
93     if (UseGarbageCollection && !UseSequenceMarker)
94     {
95         throw new NotSupportedException("To use garbage collection
96         ↪ UseSequenceMarker option must be on.");
97     }
98 }
99 }
100 }

```

./Sequences/UnicodeMap.cs

```

1  [texttt{
2  using System;
3  using System.Collections.Generic;
4  using System.Globalization;
5  using System.Runtime.CompilerServices;
6  using System.Text;
7  using Platform.Data.Sequences;
8
9  namespace Platform.Data.Doublets.Sequences
10 {
11     public class UnicodeMap
12     {
13         public static readonly ulong FirstCharLink = 1;
14         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
15         public static readonly ulong MapSize = 1 + char.MaxValue;
16
17         private readonly ILinks<ulong> _links;
18         private bool _initialized;
19
20         public UnicodeMap(ILinks<ulong> links) => _links = links;
21
22         public static UnicodeMap InitNew(ILinks<ulong> links)
23         {
24             var map = new UnicodeMap(links);
25             map.Init();
26             return map;
27         }
28
29         public void Init()
30         {
31             if (_initialized)
32             {
33                 return;
34             }
35             _initialized = true;
36             var firstLink = _links.CreatePoint();
37             if (firstLink != FirstCharLink)
38             {
39                 _links.Delete(firstLink);
40             }
41             else
42             {
43                 for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
44                 {
45                     // From NIL to It (NIL -> Character) transformation meaning, (or infinite
46                     ↪ amount of NIL characters before actual Character)
47                     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,
86             ↪ element =>
87             {
88                 sb.Append(FromLinkToChar(element));
89                 return true;
90             });
91     }
92     return sb.ToString();
93 }
94
95 public static ulong[] FromCharsToLinkArray(char[] chars) =>
96     ↪ FromCharsToLinkArray(chars, chars.Length);
97
98 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
99 {
100     // char array to ulong array
101     var linksSequence = new ulong[count];
102     for (var i = 0; i < count; i++)
103     {
104         linksSequence[i] = FromCharToLink(chars[i]);
105     }
106     return linksSequence;
107 }

```

```

108 public static ulong[] FromStringToLinkArray(string sequence)
109 {
110     // char array to ulong array
111     var linksSequence = new ulong[sequence.Length];
112     for (var i = 0; i < sequence.Length; i++)
113     {
114         linksSequence[i] = FromCharToLink(sequence[i]);
115     }
116     return linksSequence;
117 }
118
119 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
120 {
121     var result = new List<ulong[]>();
122     var offset = 0;
123     while (offset < sequence.Length)
124     {
125         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
126         var relativeLength = 1;
127         var absoluteLength = offset + relativeLength;
128         while (absoluteLength < sequence.Length &&
129             currentCategory ==
130             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
131         {
132             relativeLength++;
133             absoluteLength++;
134         }
135         // char array to ulong array
136         var innerSequence = new ulong[relativeLength];
137         var maxLength = offset + relativeLength;
138         for (var i = offset; i < maxLength; i++)
139         {
140             innerSequence[i - offset] = FromCharToLink(sequence[i]);
141         }
142         result.Add(innerSequence);
143         offset += relativeLength;
144     }
145     return result;
146 }
147
148 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
149 {
150     var result = new List<ulong[]>();
151     var offset = 0;
152     while (offset < array.Length)
153     {
154         var relativeLength = 1;
155         if (array[offset] <= LastCharLink)
156         {
157             var currentCategory =
158             ↪ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
159             var absoluteLength = offset + relativeLength;
160             while (absoluteLength < array.Length &&
161                 array[absoluteLength] <= LastCharLink &&
162                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkT
163                 ↪ oChar(array[absoluteLength])))
164             {
165                 relativeLength++;
166                 absoluteLength++;
167             }
168         }
169         else

```

```

167     {
168         var absoluteLength = offset + relativeLength;
169         while (absoluteLength < array.Length && array[absoluteLength] >
170             ↪ LastCharLink)
171         {
172             relativeLength++;
173             absoluteLength++;
174         }
175         // copy array
176         var innerSequence = new ulong[relativeLength];
177         var maxLength = offset + relativeLength;
178         for (var i = offset; i < maxLength; i++)
179         {
180             innerSequence[i - offset] = array[i];
181         }
182         result.Add(innerSequence);
183         offset += relativeLength;
184     }
185     return result;
186 }
187 }
188 }
189 }

```

./Sequences/Walkers/LeftSequenceWalker.cs

```

1  [texttt{
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Data.Doublets.Sequences.Walkers
6  {
7      public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
8      {
9          public LeftSequenceWalker(ILinks<TLink> links) : base(links) { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override IList<TLink> GetNextElementAfterPop(IList<TLink> element)
13              ↪ => Links.GetLink(Links.GetSource(element));
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected override IList<TLink> GetNextElementAfterPush(IList<TLink> element)
17              ↪ => Links.GetLink(Links.GetTarget(element));
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override IEnumerable<IList<TLink>> WalkContents(IList<TLink>
21              ↪ element)
22          {
23              var start = Links.Constants.IndexPart + 1;
24              for (var i = element.Count - 1; i >= start; i--)
25              {
26                  var partLink = Links.GetLink(element[i]);
27                  if (IsElement(partLink))
28                  {
29                      yield return partLink;
30                  }
31              }
32          }
33      }
34  }
35  }

```

./Sequences/Walkers/RightSequenceWalker.cs

```
1  [texttt{
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Data.Doublets.Sequences.Walkers
6  {
7      public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
8      {
9          public RightSequenceWalker(ILinks<TLink> links) : base(links) { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override IList<TLink> GetNextElementAfterPop(IList<TLink> element)
13              ↪ => Links.GetLink(Links.GetTarget(element));
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected override IList<TLink> GetNextElementAfterPush(IList<TLink> element)
17              ↪ => Links.GetLink(Links.GetSource(element));
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override IEnumerable<IList<TLink>> WalkContents(IList<TLink>
21              ↪ element)
22          {
23              for (var i = Links.Constants.IndexPart + 1; i < element.Count; i++)
24              {
25                  var partLink = Links.GetLink(element[i]);
26                  if (IsElement(partLink))
27                  {
28                      yield return partLink;
29                  }
30              }
31          }
32      }
33  }
```

./Sequences/Walkers/SequenceWalkerBase.cs

```
1  [texttt{
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Data.Sequences;
5
6  namespace Platform.Data.Doublets.Sequences.Walkers
7  {
8      public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
9          ↪ ISequenceWalker<TLink>
10      {
11          // TODO: Use IStack inead of System.Collections.Generic.Stack, but IStack should
12          ↪ contain IsEmpty property
13          private readonly Stack<IList<TLink>> _stack;
14
15          protected SequenceWalkerBase(ILinks<TLink> links) : base(links) => _stack = new
16          ↪ Stack<IList<TLink>>();
17
18          public IEnumerable<IList<TLink>> Walk(TLink sequence)
19          {
20              if (_stack.Count > 0)
21              {
22                  _stack.Clear(); // This can be replaced with while(!_stack.IsEmpty)
23                  ↪ _stack.Pop()
24              }
25              var element = Links.GetLink(sequence);
```

```
22          if (IsElement(element))
23          {
24              yield return element;
25          }
26          else
27          {
28              while (true)
29              {
30                  if (IsElement(element))
31                  {
32                      if (_stack.Count == 0)
33                      {
34                          break;
35                      }
36                      element = _stack.Pop();
37                      foreach (var output in WalkContents(element))
38                      {
39                          yield return output;
40                      }
41                      element = GetNextElementAfterPop(element);
42                  }
43                  else
44                  {
45                      stack.Push(element);
46                      element = GetNextElementAfterPush(element);
47                  }
48              }
49          }
50      }
51
52      [MethodImpl(MethodImplOptions.AggressiveInlining)]
53      protected virtual bool IsElement(IList<TLink> elementLink) =>
54          ↪ Point<TLink>.IsPartialPointUnchecked(elementLink);
55
56      [MethodImpl(MethodImplOptions.AggressiveInlining)]
57      protected abstract IList<TLink> GetNextElementAfterPop(IList<TLink> element);
58
59      [MethodImpl(MethodImplOptions.AggressiveInlining)]
60      protected abstract IList<TLink> GetNextElementAfterPush(IList<TLink> element);
61
62      [MethodImpl(MethodImplOptions.AggressiveInlining)]
63      protected abstract IEnumerable<IList<TLink>> WalkContents(IList<TLink>
64          ↪ element);
65  }
```

./Stacks/Stack.cs

```
1  [texttt{
2  using System.Collections.Generic;
3  using Platform.Collections.Stacks;
4
5  namespace Platform.Data.Doublets.Stacks
6  {
7      public class Stack<TLink> : IStack<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↪ EqualityComparer<TLink>.Default;
11
12          private readonly ILinks<TLink> _links;
13          private readonly TLink _stack;
```

```

14     public Stack(ILinks<TLink> links, TLink stack)
15     {
16         _links = links;
17         _stack = stack;
18     }
19
20     private TLink GetStackMarker() => _links.GetSource(_stack);
21
22     private TLink GetTop() => _links.GetTarget(_stack);
23
24     public TLink Peek() => _links.GetTarget(GetTop());
25
26     public TLink Pop()
27     {
28         var element = Peek();
29         if (!_equalityComparer.Equals(element, _stack))
30         {
31             var top = GetTop();
32             var previousTop = _links.GetSource(top);
33             _links.Update(_stack, GetStackMarker(), previousTop);
34             _links.Delete(top);
35         }
36         return element;
37     }
38
39     public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
40         ↪ _links.GetOrCreate(GetTop(), element));
41 }
42 }

```

./Stacks/StackExtensions.cs

```

1  namespace Platform.Data.Doublets.Stacks
2  {
3      public static class StackExtensions
4      {
5          public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink
6              ↪ stackMarker)
7          {
8              var stackPoint = links.CreatePoint();
9              var stack = links.Update(stackPoint, stackMarker, stackPoint);
10             return stack;
11         }
12
13         public static void DeleteStack<TLink>(this ILinks<TLink> links, TLink stack) =>
14             ↪ links.Delete(stack);
15     }
16 }

```

./SynchronizedLinks.cs

```

1  namespace Platform.Data.Doublets
2  {
3      /// <remarks>

```

```

11     /// TODO: Autogeneration of synchronized wrapper (decorator).
12     /// TODO: Try to unfold code of each method using IL generation for performance
13     ↪ improvements.
14     /// TODO: Or even to unfold multiple layers of implementations.
15     /// </remarks>
16     public class SynchronizedLinks<T> : ISynchronizedLinks<T>
17     {
18         public LinksCombinedConstants<T, T, int> Constants { get; }
19         public ISynchronization SyncRoot { get; }
20         public ILinks<T> Sync { get; }
21         public ILinks<T> Unsync { get; }
22
23         public SynchronizedLinks(ILinks<T> links) : this(new
24             ↪ ReaderWriterLockSynchronization(), links) { }
25
26         public SynchronizedLinks(ISynchronization synchronization, ILinks<T> links)
27         {
28             SyncRoot = synchronization;
29             Sync = this;
30             Unsync = links;
31             Constants = links.Constants;
32         }
33
34         public T Count(IList<T> restriction) =>
35             ↪ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
36         public T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
37             ↪ SyncRoot.ExecuteReadOperation(handler, restrictions, (handler1, restrictions1)
38                 ↪ => Unsync.Each(handler1, restrictions1));
39         public T Create() => SyncRoot.ExecuteWriteOperation(Unsync.Create);
40         public T Update(IList<T> restrictions) =>
41             ↪ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Update);
42         public void Delete(T link) => SyncRoot.ExecuteWriteOperation(link, Unsync.Delete);
43
44         //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T>
45         ↪ matchedHandler, IList<T> substitution, Func<IList<T>, IList<T>, T>
46         ↪ substitutedHandler)
47         // {
48         //     if (restriction != null && substitution != null &&
49         ↪ !substitution.EqualTo(restriction))
50         //         return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
51             ↪ substitution, substitutedHandler, Unsync.Trigger);
52         //     return SyncRoot.ExecuteReadOperation(restriction, matchedHandler,
53             ↪ substitution, substitutedHandler, Unsync.Trigger);
54         // }
55     }
56 }

```

./UInt64Link.cs

```

1  namespace Platform.Data.Doublets
2  {
3      using System;
4      using System.Collections;
5      using System.Collections.Generic;
6      using Platform.Exceptions;
7      using Platform.Ranges;
8      using Platform.Helpers.Singletons;
9      using Platform.Data.Constants;
10
11     namespace Platform.Data.Doublets
12     {
13         /// <summary>

```



```

13  /// Структура описывающая уникальную связь.
14  /// </summary>
15  public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>,
16  ↪ IList<ulong>
17  {
18      private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
19      ↪ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
20
21      private const int Length = 3;
22
23      public readonly ulong Index;
24      public readonly ulong Source;
25      public readonly ulong Target;
26
27      public static readonly UInt64Link Null = new UInt64Link();
28
29      public UInt64Link(params ulong[] 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 UInt64Link(IList<ulong> 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 UInt64Link(ulong index, ulong source, ulong target)
50      {
51          Index = index;
52          Source = source;
53          Target = target;
54      }
55
56      public UInt64Link(ulong source, ulong target)
57      : this(_constants.Null, source, target)
58      {
59          Source = source;
60          Target = target;
61      }
62
63      public static UInt64Link Create(ulong source, ulong target) => new
64      ↪ UInt64Link(source, target);
65
66      public override int GetHashCode() => (Index, Source, Target).GetHashCode();
67
68      public bool IsNull() => Index == _constants.Null
69      ↪ && Source == _constants.Null
70      ↪ && Target == _constants.Null;
71
72      public override bool Equals(object other) => other is UInt64Link &&
73      ↪ Equals((UInt64Link)other);
74
75      public bool Equals(UInt64Link other) => Index == other.Index
76      ↪ && Source == other.Source

```

```

67      && Target == other.Target;
68
69      public static string ToString(ulong index, ulong source, ulong target) => $"({index}:
70      ↪ {source}->{target})";
71
72      public static string ToString(ulong source, ulong target) => $"({source}->{target})";
73
74      public static implicit operator ulong[(UInt64Link link) => link.ToArray();
75
76      public static implicit operator UInt64Link(ulong[] linkArray) => new
77      ↪ UInt64Link(linkArray);
78
79      public ulong[] ToArray()
80      {
81          var array = new ulong[Length];
82          CopyTo(array, 0);
83          return array;
84      }
85
86      public override string ToString() => Index == _constants.Null ? ToString(Source,
87      ↪ Target) : ToString(Index, Source, Target);
88
89      #region IList
90
91      public ulong this[int index]
92      {
93          get
94          {
95              Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
96              ↪ 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
110              ↪ Ensure.ArgumentInRange
111          }
112          set => throw new NotSupportedException();
113      }
114
115      public int Count => Length;
116
117      public bool IsReadOnly => true;
118
119      IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
120
121      public IEnumerator<ulong> GetEnumerator()
122      {
123          yield return Index;
124          yield return Source;
125          yield return Target;
126      }
127
128      public void Add(ulong item) => throw new NotSupportedException();

```

```

125 public void Clear() => throw new NotSupportedException();
126
127 public bool Contains(ulong item) => IndexOf(item) >= 0;
128
129 public void CopyTo(ulong[] array, int arrayIndex)
130 {
131     Ensure.Always.ArgumentNotNull(array, nameof(array));
132     Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length -
133         ↪ 1), nameof(arrayIndex));
134     if (arrayIndex + Length > array.Length)
135     {
136         throw new ArgumentException();
137     }
138     array[arrayIndex++] = Index;
139     array[arrayIndex++] = Source;
140     array[arrayIndex] = Target;
141 }
142
143 public bool Remove(ulong item) =>
144     ↪ Throw.A.NotSupportedExceptionAndReturn<bool>();
145
146 public int IndexOf(ulong item)
147 {
148     if (Index == item)
149     {
150         return _constants.IndexPart;
151     }
152     if (Source == item)
153     {
154         return _constants.SourcePart;
155     }
156     if (Target == item)
157     {
158         return _constants.TargetPart;
159     }
160     return -1;
161 }
162
163 public void Insert(int index, ulong item) => throw new NotSupportedException();
164
165 public void RemoveAt(int index) => throw new NotSupportedException();
166
167 #endregion
168 }
169 }

```

./UInt64LinkExtensions.cs

```

1  [texttt{
2  namespace Platform.Data.Doublets
3  {
4      public static class UInt64LinkExtensions
5      {
6          public static bool IsFullPoint(this UInt64Link link) =>
7              ↪ Point<ulong>.IsFullPoint(link);
8          public static bool IsPartialPoint(this UInt64Link link) =>
9              ↪ Point<ulong>.IsPartialPoint(link);
10     }
11 }

```

./UInt64LinksExtensions.cs

```

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

```

```

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

```

```

108         var source = new UInt64Link(links.GetLink(link.Source));
109         if (isElement(source))
110         {
111             appendElement(sb, source);
112         }
113         else
114         {
115             links.AppendStructure(sb, visited, source.Index, isElement,
↪ appendElement, renderIndex);
116         }
117     }
118     sb.Append(' ');
119     if (link.Target == link.Index)
120     {
121         sb.Append(link.Index);
122     }
123     else
124     {
125         var target = new UInt64Link(links.GetLink(link.Target));
126         if (isElement(target))
127         {
128             appendElement(sb, target);
129         }
130         else
131         {
132             links.AppendStructure(sb, visited, target.Index, isElement,
↪ appendElement, renderIndex);
133         }
134     }
135     sb.Append(')');
136 }
137 else
138 {
139     if (renderDebug)
140     {
141         sb.Append('*');
142     }
143     sb.Append(linkIndex);
144 }
145 }
146 else
147 {
148     if (renderDebug)
149     {
150         sb.Append('~');
151     }
152     sb.Append(linkIndex);
153 }
154 }
155 }
156 }
157 }

```

./UInt64LinksTransactionsLayer.cs

```

1  namespace textttt{
2      using System;
3      using System.Linq;
4      using System.Collections.Generic;
5      using System.IO;
6      using System.Runtime.CompilerServices;
7      using System.Threading;

```

```

8 using System.Threading.Tasks;
9 using Platform.Disposables;
10 using Platform.Timestamps;
11 using Platform.Unsafe;
12 using Platform.IO;
13 using Platform.Data.Doublets.Decorators;
14
15 namespace Platform.Data.Doublets
16 {
17     public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong>
18     ↪     ↪ // -V3073
19     {
20         // <remarks>
21         // Альтернативные варианты хранения трансформации (элемента транзакции):
22         //
23         // private enum TransitionType
24         // {
25         //     Creation,
26         //     UpdateOf,
27         //     UpdateTo,
28         //     Deletion
29         // }
30         // private struct Transition
31         // {
32         //     public ulong TransactionId;
33         //     public UniqueTimestamp Timestamp;
34         //     public TransactionItemType Type;
35         //     public Link Source;
36         //     public Link Linker;
37         //     public Link Target;
38         // }
39         // Или
40         // public struct TransitionHeader
41         // {
42         //     public ulong TransactionIdCombined;
43         //     public ulong TimestampCombined;
44         //
45         //     public ulong TransactionId
46         //     {
47         //         get
48         //         {
49         //             return (ulong) mask & TransactionIdCombined;
50         //         }
51         //     }
52         //
53         //     public UniqueTimestamp Timestamp
54         //     {
55         //         get
56         //         {
57         //             return (UniqueTimestamp) mask & TransactionIdCombined;
58         //         }
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

```

```

69         throw new NotImplementedException();
70     }
71 }
72 }
73
74 private struct Transition
75 {
76     public TransitionHeader Header;
77     public Link Source;
78     public Link Linker;
79     public Link Target;
80 }
81
82 // </remarks>
83 public struct Transition
84 {
85     public static readonly long Size = StructureHelpers.SizeOf<Transition>();
86
87     public readonly ulong TransactionId;
88     public readonly UInt64Link Before;
89     public readonly UInt64Link After;
90     public readonly Timestamp Timestamp;
91
92     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
93     ↪     ↪ transactionId, UInt64Link before, UInt64Link after)
94     {
95         TransactionId = transactionId;
96         Before = before;
97         After = after;
98         Timestamp = uniqueTimestampFactory.Create();
99     }
100
101     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
102     ↪     ↪ transactionId, UInt64Link before)
103     : this(uniqueTimestampFactory, transactionId, before, default)
104     {
105     }
106
107     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
108     ↪     ↪ transactionId)
109     : this(uniqueTimestampFactory, transactionId, default, default)
110     {
111     }
112
113     public override string ToString() => $"{{Timestamp}} {{TransactionId}}: {{Before}}
114     ↪     ↪ {{After}}";
115 }
116
117 /// <remarks>
118 /// Другие варианты реализации транзакций (атомарности):
119 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
120 ///     ↪ Target)) и индексов.
121 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но
122 ///     ↪ дополнительно потребуется решить вопрос
123 ///     ↪ со ссылками на внешние идентификаторы, или как-то иначе решить
124 ///     ↪ вопрос с пересечениями идентификаторов.
125
126 /// Где хранить промежуточный список транзакций?
127
128 /// В оперативной памяти:
129 /// Минусы:

```

```

123  /// 1. Может усложнить систему, если она будет функционировать
124  ↪ самостоятельно,
125  /// так как нужно отдельно выделять память под список трансформаций.
126  /// 2. Выделенной оперативной памяти может не хватить, в том случае,
127  /// если транзакция использует слишком много трансформаций.
128  /// -> Можно использовать жёсткий диск для слишком длинных транзакций.
129  ↪ -> Максимальный размер списка трансформаций можно ограничить /
130  ↪ задать константой.
131  /// 3. При подтверждении транзакции (Commit) все трансформации
132  ↪ записываются разом создавая задержку.
133  ///
134  /// На жёстком диске:
135  /// Минусы:
136  /// 1. Длительный отклик, на запись каждой трансформации.
137  /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
138  /// -> Это может решаться упаковкой/исключением дублирующих операций.
139  /// -> Также это может решаться тем, что короткие транзакции вообще
140  /// не будут записываться в случае отката.
141  /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться
142  ↪ пока все операции (трансформации)
143  ↪ будут записаны в лог.
144  /// </remarks>
145  public class Transaction : DisposableBase
146  {
147  private readonly Queue<Transition> _transitions;
148  private readonly UInt64LinksTransactionsLayer _layer;
149  public bool IsCommitted { get; private set; }
150  public bool IsReverted { get; private set; }
151
152  public Transaction(UInt64LinksTransactionsLayer layer)
153  {
154  layer = layer;
155  if (_layer._currentTransactionId != 0)
156  {
157  throw new NotSupportedException("Nested transactions not supported.");
158  }
159  IsCommitted = false;
160  IsReverted = false;
161  transitions = new Queue<Transition>();
162  SetCurrentTransaction(layer, this);
163  }
164
165  public void Commit()
166  {
167  EnsureTransactionAllowsWriteOperations(this);
168  while (_transitions.Count > 0)
169  {
170  var transition = _transitions.Dequeue();
171  _layer._transitions.Enqueue(transition);
172  }
173  layer._lastCommittedTransactionId = _layer._currentTransactionId;
174  IsCommitted = true;
175  }
176
177  private void Revert()
178  {
179  EnsureTransactionAllowsWriteOperations(this);
180  var transitionsToRevert = new Transition[_transitions.Count];
181  transitions.CopyTo(transitionsToRevert, 0);
182  for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
183  {
184  _layer.RevertTransition(transitionsToRevert[i]);
185  }
186  }

```

```

182  }
183  IsReverted = true;
184  }
185
186  public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
187  ↪ Transaction transaction)
188  {
189  layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
190  layer._currentTransactionTransitions = transaction._transitions;
191  layer._currentTransaction = transaction;
192  }
193
194  public static void EnsureTransactionAllowsWriteOperations(Transaction
195  ↪ transaction)
196  {
197  if (transaction.IsReverted)
198  {
199  throw new InvalidOperationException("Transation is reverted.");
200  }
201  if (transaction.IsCommitted)
202  {
203  throw new InvalidOperationException("Transation is committed.");
204  }
205  }
206
207  protected override void DisposeCore(bool manual, bool wasDisposed)
208  {
209  if (!wasDisposed && _layer != null && !_layer.IsDisposed)
210  {
211  if (!IsCommitted && !IsReverted)
212  {
213  Revert();
214  }
215  _layer.ResetCurrentTransation();
216  }
217  }
218
219  // TODO: THIS IS EXCEPTION WORKAROUND, REMOVE IT THEN
220  ↪ https://github.com/linksplatform/Disposables/issues/13 FIXED
221  protected override bool AllowMultipleDisposeCalls => true;
222  }
223
224  public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
225
226  private readonly string _logAddress;
227  private readonly FileStream _log;
228  private readonly Queue<Transition> _transitions;
229  private readonly UniqueTimestampFactory _uniqueTimestampFactory;
230  private Task _transitionsPusher;
231  private Transition _lastCommittedTransition;
232  private ulong _currentTransactionId;
233  private Queue<Transition> _currentTransactionTransitions;
234  private Transaction _currentTransaction;
235  private ulong _lastCommittedTransactionId;
236
237  public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
238  : base(links)
239  {
240  if (string.IsNullOrEmpty(logAddress))
241  {
242  throw new ArgumentNullException(nameof(logAddress));
243  }
244  }

```

```

241 // В первой строке файла хранится последняя закоммиченную транзакцию.
242 // При запуске это используется для проверки удачного закрытия файла лога.
243 // In the first line of the file the last committed transaction is stored.
244 // On startup, this is used to check that the log file is successfully closed.
245 var lastCommittedTransition =
    ↪ FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
246 var lastWrittenTransition =
    ↪ FileHelpers.ReadLastOrDefault<Transition>(logAddress);
247 if (!lastCommittedTransition.Equals(lastWrittenTransition))
248 {
249     Dispose();
250     throw new NotSupportedException("Database is damaged, autorecovery is not
    ↪ supported yet.");
251 }
252 if (lastCommittedTransition.Equals(default(Transition)))
253 {
254     FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
255 }
256 lastCommittedTransition = lastCommittedTransition;
257 // TODO: Think about a better way to calculate or store this value
258 var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
259 _lastCommittedTransactionId = allTransitions.Max(x => x.TransactionId);
260 _uniqueTimestampFactory = new UniqueTimestampFactory();
261 _logAddress = logAddress;
262 _log = FileHelpers.Append(logAddress);
263 _transitions = new Queue<Transition>();
264 _transitionsPusher = new Task(TransitionsPusher);
265 _transitionsPusher.Start();
266 }
267
268 public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
269
270 public override ulong Create()
271 {
272     var createdLinkIndex = Links.Create();
273     var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
274     CommitTransition(new Transition(_uniqueTimestampFactory,
    ↪ _currentTransactionId, default, 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, afterLink));
284     return parts[Constants.IndexPart];
285 }
286
287 public override void Delete(ulong link)
288 {
289     var deletedLink = new UInt64Link(Links.GetLink(link));
290     Links.Delete(link);
291     CommitTransition(new Transition(_uniqueTimestampFactory,
    ↪ _currentTransactionId, deletedLink, default));
292 }
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 private Queue<Transition> GetCurrentTransitions() =>
    ↪ _currentTransactionTransitions ?? _transitions;

```

```

296
297 private void CommitTransition(Transition transition)
298 {
299     if (_currentTransaction != null)
300     {
301         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
302     }
303     var transitions = GetCurrentTransitions();
304     transitions.Enqueue(transition);
305 }
306
307 private void RevertTransition(Transition transition)
308 {
309     if (transition.After.IsNull()) // Revert Deletion with Creation
310     {
311         Links.Create();
312     }
313     else if (transition.Before.IsNull()) // Revert Creation with Deletion
314     {
315         Links.Delete(transition.After.Index);
316     }
317     else // Revert Update
318     {
319         Links.Update(new[] { transition.After.Index, transition.Before.Source,
    ↪ transition.Before.Target });
320     }
321 }
322
323 private void ResetCurrentTransation()
324 {
325     _currentTransactionId = 0;
326     _currentTransactionTransitions = null;
327     _currentTransaction = null;
328 }
329
330 private void PushTransitions()
331 {
332     if (_log == null || _transitions == null)
333     {
334         return;
335     }
336     for (var i = 0; i < _transitions.Count; i++)
337     {
338         var transition = _transitions.Dequeue();
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 {
358     try
359     {
360         var pusher = _transitionsPusher;
361         if (pusher != null)
362         {
363             _transitionsPusher = null;
364             pusher.Wait();
365         }
366         if (_transitions != null)
367         {
368             PushTransitions();
369         }
370         Disposable.TryDispose(_log);
371         FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
372     }
373     catch
374     {

```

```
375     }
376 }
377
378 #region DisposalBase
379
380 protected override void DisposeCore(bool manual, bool wasDisposed)
381 {
382     if (!wasDisposed)
383     {
384         DisposeTransitions();
385     }
386     base.DisposeCore(manual, wasDisposed);
387 }
388
389 #endregion
390
391 }
392 }
```

Index

./Converters/AddressToUnaryNumberConverter.cs, 1
./Converters/LinkToltsFrequencyNumberConveter.cs, 1
./Converters/PowerOf2ToUnaryNumberConverter.cs, 1
./Converters/UnaryNumberToAddressAddOperationConverter.cs, 2
./Converters/UnaryNumberToAddressOrOperationConverter.cs, 2
./Decorators/LinksCascadeDependenciesResolver.cs, 3
./Decorators/LinksCascadeUniquenessAndDependenciesResolver.cs, 3
./Decorators/LinksDecoratorBase.cs, 3
./Decorators/LinksDependenciesValidator.cs, 4
./Decorators/LinksDisposableDecoratorBase.cs, 4
./Decorators/LinksInnerReferenceValidator.cs, 4
./Decorators/LinksNonExistentReferencesCreator.cs, 5
./Decorators/LinksNullToSelfReferenceResolver.cs, 5
./Decorators/LinksSelfReferenceResolver.cs, 5
./Decorators/LinksUniquenessResolver.cs, 6
./Decorators/LinksUniquenessValidator.cs, 6
./Decorators/NonNullContentsLinkDeletionResolver.cs, 6
./Decorators/UInt64Links.cs, 6
./Decorators/UniLinks.cs, 7
./Doublet.cs, 10
./DoubletComparer.cs, 10
./Hybrid.cs, 11
./ILinks.cs, 11
./ILinksExtensions.cs, 12
./ISynchronizedLinks.cs, 19
./Incrementers/FrequencyIncrementer.cs, 18
./Incrementers/LinkFrequencyIncrementer.cs, 18
./Incrementers/UnaryNumberIncrementer.cs, 18
./Link.cs, 19
./LinkExtensions.cs, 20
./LinksOperatorBase.cs, 20
./PropertyOperators/DefaultLinkPropertyOperator.cs, 21
./PropertyOperators/FrequencyPropertyOperator.cs, 21
./ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs, 27
./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs, 28
./ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 21
./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs, 36
./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs, 36
./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 32
./Sequences/Converters/BalancedVariantConverter.cs, 40
./Sequences/Converters/CompressingConverter.cs, 41
./Sequences/Converters/LinksListToSequenceConverterBase.cs, 43
./Sequences/Converters/OptimalVariantConverter.cs, 43
./Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 44
./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs, 44
./Sequences/CreteriaMatchers/MarkedSequenceCreteriaMatcher.cs, 44
./Sequences/DefaultSequenceAppender.cs, 45
./Sequences/DuplicateSegmentsCounter.cs, 45
./Sequences/DuplicateSegmentsProvider.cs, 45
./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs, 47
./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToltsFrequencyNumberConverter.cs, 47
./Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 47
./Sequences/Frequencies/Cache/LinkFrequency.cs, 48
./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 49
./Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 49
./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 49
./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 49
./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 50
./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 50
./Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 50
./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 51
./Sequences/HeightProviders/ISequenceHeightProvider.cs, 51
./Sequences/Sequences.Experiments.ReadSequence.cs, 74
./Sequences/Sequences.Experiments.cs, 57
./Sequences/Sequences.cs, 51
./Sequences/SequencesExtensions.cs, 75
./Sequences/SequencesIndexer.cs, 75
./Sequences/SequencesOptions.cs, 76
./Sequences/UnicodeMap.cs, 77
./Sequences/Walkers/LeftSequenceWalker.cs, 78
./Sequences/Walkers/RightSequenceWalker.cs, 78
./Sequences/Walkers/SequenceWalkerBase.cs, 79
./Stacks/Stack.cs, 79
./Stacks/StackExtensions.cs, 80
./SynchronizedLinks.cs, 80
./UInt64Link.cs, 80
./UInt64LinkExtensions.cs, 82
./UInt64LinksExtensions.cs, 82
./UInt64LinksTransactionsLayer.cs, 83
./obj/Debug/netstandard2.0/Platform.Data.Doublets.AssemblyInfo.cs, 20