

# LinksPlatform's Platform.Data.Doublets Class Library

## ./Platform.Data.Doublets/Converters/AddressToUnaryNumberConverter.cs

```

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

```

## ./Platform.Data.Doublets/Converters/LinkToItsFrequencyNumberConveter.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Converters
6  {
7      public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
8          ⇨ IConverter<Doublet<TLink>, TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ⇨ EqualityComparer<TLink>.Default;
12
13         private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
14         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
15
16         public LinkToItsFrequencyNumberConveter(
17             ILinks<TLink> links,
18             IPropertyOperator<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 ({doublet}) not found.", nameof(doublet));
32             }
33             var frequency = _frequencyPropertyOperator.Get(link);
34             if (_equalityComparer.Equals(frequency, default))
35             {

```

```

34         return default;
35     }
36     var frequencyNumber = Links.GetSource(frequency);
37     return _unaryNumberToAddressConverter.Convert(frequencyNumber);
38 }
39 }
40 }

```

./Platform.Data.Doublets/Converters/PowerOf2ToUnaryNumberConverter.cs

```

1  using System.Collections.Generic;
2  using Platform.Exceptions;
3  using Platform.Interfaces;
4  using Platform.Ranges;
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) : base(links)
17         {
18             _unaryNumberPowersOf2 = new TLink[64];
19             _unaryNumberPowersOf2[0] = one;
20         }
21
22         public TLink Convert(int power)
23         {
24             Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
25                 ⇨ - 1), nameof(power));
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 }

```

./Platform.Data.Doublets/Converters/UnaryNumberToAddressAddOperationConverter.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Numbers;
5
6  namespace Platform.Data.Doublets.Converters
7  {
8      public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
9          ⇨ 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 unaryOne)
18             : base(links)
19         {
20             _unaryOne = unaryOne;
21             InitUnaryToUInt64();
22         }
23
24         private void InitUnaryToUInt64()
25         {
26             var one = Integer<TLink>.One;
27             _unaryToUInt64 = new Dictionary<TLink, TLink>
28             {
29                 { _unaryOne, one }
30             };
31             var unary = _unaryOne;
32             var number = one;
33             for (var i = 1; i < 64; i++)

```

```

32     {
33         unary = Links.GetOrCreate(unary, unary);
34         number = Double(number);
35         _unaryToUInt64.Add(unary, number);
36     }
37 }
38
39 public TLink Convert(TLink unaryNumber)
40 {
41     if (_equalityComparer.Equals(unaryNumber, default))
42     {
43         return default;
44     }
45     if (_equalityComparer.Equals(unaryNumber, _unaryOne))
46     {
47         return Integer<TLink>.One;
48     }
49     var source = Links.GetSource(unaryNumber);
50     var target = Links.GetTarget(unaryNumber);
51     if (_equalityComparer.Equals(source, target))
52     {
53         return _unaryToUInt64[unaryNumber];
54     }
55     else
56     {
57         var result = _unaryToUInt64[source];
58         TLink lastValue;
59         while (!_unaryToUInt64.TryGetValue(target, out lastValue))
60         {
61             source = Links.GetSource(target);
62             result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
63             target = Links.GetTarget(target);
64         }
65         result = Arithmetic<TLink>.Add(result, lastValue);
66         return result;
67     }
68 }
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 private static TLink Double(TLink number) => (Integer<TLink>)((Integer<TLink>)number *
72     ↪ 2UL);
73 }

```

#### ./Platform.Data.Doublets/Converters/UnaryNumberToAddressOrOperationConverter.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Reflection;
4  using Platform.Numbers;
5  using System.Runtime.CompilerServices;
6
7  namespace Platform.Data.Doublets.Converters
8  {
9      public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
10         ↪ 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, IConverter<int,
18             ↪ TLink> powerOf2ToUnaryNumberConverter)
19             : base(links)
20         {
21             _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
22             for (int i = 0; i < Type<TLink>.BitsLength; i++)
23             {
24                 _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
25             }
26         }
27
28         public TLink Convert(TLink sourceNumber)
29         {
30             var nullConstant = Links.Constants.Null;
31             var source = sourceNumber;
32             var target = nullConstant;
33             if (!_equalityComparer.Equals(source, nullConstant))
34             {

```

```

32         while (true)
33         {
34             if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
35             {
36                 SetBit(ref target, powerOf2Index);
37                 break;
38             }
39             else
40             {
41                 powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
42                 SetBit(ref target, powerOf2Index);
43                 source = Links.GetTarget(source);
44             }
45         }
46     }
47     return target;
48 }
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 private static void SetBit(ref TLink target, int powerOf2Index) => target =
    ↪ (Integer<TLink>)((Integer<TLink>)target | 1UL << powerOf2Index); // Should be
    ↪ Math.Or(target, Math.ShiftLeft(One, powerOf2Index))
52 }
53 }

```

./Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs

```

1 namespace Platform.Data.Doublets.Decorators
2 {
3     public class LinksCascadeUniquenessAndUsagesResolver<TLink> : LinksUniquenessResolver<TLink>
4     {
5         public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLink> links) : base(links) { }
6
7         protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
            ↪ newLinkAddress)
8         {
9             Links.MergeUsages(oldLinkAddress, newLinkAddress);
10            return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
11        }
12    }
13 }

```

./Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs

```

1 namespace Platform.Data.Doublets.Decorators
2 {
3     /// <remarks>
4     /// <para>Must be used in conjunction with NonNullContentsLinkDeletionResolver.</para>
5     /// <para>Должен использоваться вместе с NonNullContentsLinkDeletionResolver.</para>
6     /// </remarks>
7     public class LinksCascadeUsagesResolver<TLink> : LinksDecoratorBase<TLink>
8     {
9         public LinksCascadeUsagesResolver(ILinks<TLink> links) : base(links) { }
10
11         public override void Delete(TLink linkIndex)
12         {
13             this.DeleteAllUsages(linkIndex);
14             Links.Delete(linkIndex);
15         }
16     }
17 }

```

./Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Data.Constants;
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
8     {
9         public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
10        protected LinksDecoratorBase(ILinks<TLink> links) : base(links) => Constants =
            ↪ links.Constants;
11        public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
12        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
            ↪ => Links.Each(handler, restrictions);
13        public virtual TLink Create() => Links.Create();
14        public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
15        public virtual void Delete(TLink link) => Links.Delete(link);

```

```

16     }
17 }

```

./Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Disposables;
4  using Platform.Data.Constants;
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public abstract class LinksDisposableDecoratorBase<TLink> : DisposableBase, ILinks<TLink>
9      {
10         public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
11
12         public ILinks<TLink> Links { get; }
13
14         protected LinksDisposableDecoratorBase(ILinks<TLink> links)
15         {
16             Links = links;
17             Constants = links.Constants;
18         }
19
20         public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
21
22         public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
23             => Links.Each(handler, restrictions);
24
25         public virtual TLink Create() => Links.Create();
26
27         public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
28
29         public virtual void Delete(TLink link) => Links.Delete(link);
30
31         protected override bool AllowMultipleDisposeCalls => true;
32
33         protected override void Dispose(bool manual, bool wasDisposed)
34         {
35             if (!wasDisposed)
36             {
37                 Links.DisposeIfPossible();
38             }
39         }
40     }

```

./Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets.Decorators
5  {
6      // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
7      // be external (hybrid link's raw number).
8      public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
9      {
10         public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
11
12         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
13         {
14             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
15             return Links.Each(handler, restrictions);
16         }
17
18         public override TLink Count(IList<TLink> restriction)
19         {
20             Links.EnsureInnerReferenceExists(restriction, nameof(restriction));
21             return Links.Count(restriction);
22         }
23
24         public override TLink Update(IList<TLink> restrictions)
25         {
26             // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
27             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
28             return Links.Update(restrictions);
29         }
30
31         public override void Delete(TLink link)
32         {
33             Links.EnsureLinkExists(link, nameof(link));

```

```

33         Links.Delete(link);
34     }
35 }
36 }

```

#### ./Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets.Decorators
5  {
6      public class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
7      {
8          private static readonly EqualityComparer<TLink> _equalityComparer =
9              ↳ EqualityComparer<TLink>.Default;
10
11          public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
12
13          public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
14          {
15              var constants = Constants;
16              var itselfConstant = constants.Itself;
17              var indexPartConstant = constants.IndexPart;
18              var sourcePartConstant = constants.SourcePart;
19              var targetPartConstant = constants.TargetPart;
20              var restrictionsCount = restrictions.Count;
21              if (!_equalityComparer.Equals(constants.Any, itselfConstant)
22                  && ((restrictionsCount > indexPartConstant) &&
23                      ↳ _equalityComparer.Equals(restrictions[indexPartConstant], itselfConstant))
24                  || ((restrictionsCount > sourcePartConstant) &&
25                      ↳ _equalityComparer.Equals(restrictions[sourcePartConstant], itselfConstant))
26                  || ((restrictionsCount > targetPartConstant) &&
27                      ↳ _equalityComparer.Equals(restrictions[targetPartConstant], itselfConstant))))
28              {
29                  // Itself constant is not supported for Each method right now, skipping execution
30                  return constants.Continue;
31              }
32              return Links.Each(handler, restrictions);
33          }
34
35          public override TLink Update(IList<TLink> restrictions) =>
36              ↳ Links.Update(Links.ResolveConstantAsSelfReference(Constants.Itself, restrictions));
37      }
38 }

```

#### ./Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs

```

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

```

#### ./Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs

```

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

```

```

8
9     public override TLink Create()
10    {
11        var link = Links.Create();
12        return Links.Update(link, link, link);
13    }
14
15    public override TLink Update(IList<TLink> restrictions) =>
16        ↪ Links.Update(Links.ResolveConstantAsSelfReference(Constants.Null, restrictions));
17 }

```

./Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs

```

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

```

./Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs

```

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

```

./Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs

```

1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      public class LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
6      {
7          public LinksUsagesValidator(ILinks<TLink> links) : base(links) { }
8
9          public override TLink Update(IList<TLink> restrictions)
10         {

```

```

11         Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
12         return Links.Update(restrictions);
13     }
14
15     public override void Delete(TLink link)
16     {
17         Links.EnsureNoUsages(link);
18         Links.Delete(link);
19     }
20 }
21 }

```

#### ./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs

```

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

```

#### ./Platform.Data.Doublets/Decorators/UInt64Links.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Collections;
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     /// <summary>
8     /// Представляет объект для работы с базой данных (файлом) в формате Links (массива связей).
9     /// </summary>
10    /// <remarks>
11    /// Возможные оптимизации:
12    /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
13    ///     + меньше объём БД
14    ///     - меньше производительность
15    ///     - больше ограничение на количество связей в БД)
16    /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
17    ///     + меньше объём БД
18    ///     - больше сложность
19    ///
20    /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
21    ///     ↳ поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
22    ///     ↳ 460 752 303 423 488
23    /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
24    ///     ↳ (битовыми строками) - вариант матрицы (выстраиваемой лениво).
25    ///
26    /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
27    ///     ↳ выбрасываться только при #if DEBUG
28    /// </remarks>
29    public class UInt64Links : LinksDisposableDecoratorBase<ulong>
30    {
31        public UInt64Links(ILinks<ulong> links) : base(links) { }
32
33        public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
34        {
35            this.EnsureLinkIsAnyOrExists(restrictions);
36            return Links.Each(handler, restrictions);
37        }
38
39        public override ulong Create() => Links.CreatePoint();
40
41        public override ulong Update(IList<ulong> restrictions)
42        {
43            var constants = Constants;
44            var nullConstant = constants.Null;
45            if (restrictions.IsNullOrEmpty())
46            {
47                return nullConstant;
48            }
49            // TODO: Looks like this is a common type of exceptions linked with restrictions
50            //     ↳ support
51            if (restrictions.Count != 3)

```



```

47     {
48         throw new NotSupportedException();
49     }
50     var indexPartConstant = constants.IndexPart;
51     var updatedLink = restrictions[indexPartConstant];
52     this.EnsureLinkExists(updatedLink,
53         ↪ $"{nameof(restrictions)}[{nameof(indexPartConstant)}]");
54     var sourcePartConstant = constants.SourcePart;
55     var newSource = restrictions[sourcePartConstant];
56     this.EnsureLinkIsItselfOrExists(newSource,
57         ↪ $"{nameof(restrictions)}[{nameof(sourcePartConstant)}]");
58     var targetPartConstant = constants.TargetPart;
59     var newTarget = restrictions[targetPartConstant];
60     this.EnsureLinkIsItselfOrExists(newTarget,
61         ↪ $"{nameof(restrictions)}[{nameof(targetPartConstant)}]");
62     var existedLink = nullConstant;
63     var itselfConstant = constants.Itself;
64     if (newSource != itselfConstant && newTarget != itselfConstant)
65     {
66         existedLink = this.SearchOrDefault(newSource, newTarget);
67     }
68     if (existedLink == nullConstant)
69     {
70         var before = Links.GetLink(updatedLink);
71         if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
72             ↪ newTarget)
73         {
74             Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
75                 ↪ newSource,
76                 newTarget == itselfConstant ? updatedLink :
77                     ↪ newTarget);
78         }
79         return updatedLink;
80     }
81     else
82     {
83         return this.MergeAndDelete(updatedLink, existedLink);
84     }
85 }
86
87 public override void Delete(ulong linkIndex)
88 {
89     Links.EnsureLinkExists(linkIndex);
90     Links.EnforceResetValues(linkIndex);
91     this.DeleteAllUsages(linkIndex);
92     Links.Delete(linkIndex);
93 }
94 }
95 }

```

./Platform.Data.Doublets/Decorators/UniLinks.cs

```

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

```

```

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

```

```

90         ///};
91         ///         if (!Memory.Each(handler, restriction))
92         ///             return Constants.Break;
93         ///     }
94         /// }
95         /// else
96         /// {
97         ///     if (substitution != null)
98         ///     {
99         ///         transitions = new List<IList<T>>();
100        ///         Func<T, bool> handler = link =>
101        ///         {
102        ///             var matchedLink = Memory.GetLinkValue(link);
103        ///             transitions.Add(matchedLink);
104        ///             return true;
105        ///         };
106        ///         if (!Memory.Each(handler, restriction))
107        ///             return Constants.Break;
108        ///     }
109        ///     else
110        ///     {
111        ///         return Constants.Continue;
112        ///     }
113        /// }
114        ///}
115        ///if (substitution != null)
116        ///{
117        ///    // Есть причина делать замену (запись)
118        ///    if (substitutedHandler != null)
119        ///    {
120        ///    }
121        ///    else
122        ///    {
123        ///    }
124        ///}
125        ///return Constants.Continue;
126
127        //if (restriction.IsNullOrEmpty()) // Create
128        //{
129        //    substitution[Constants.IndexPart] = Memory.AllocateLink();
130        //    Memory.SetLinkValue(substitution);
131        //}
132        //else if (substitution.IsNullOrEmpty()) // Delete
133        //{
134        //    Memory.FreeLink(restriction[Constants.IndexPart]);
135        //}
136        //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
137        //{
138        //    // No need to collect links to list
139        //    // Skip == Continue
140        //    // No need to check substitutedHandler
141        //    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
142        //        ↪ Constants.Break), restriction))
143        //        return Constants.Break;
144        //}
145        //else // Update
146        //{
147        //    //List<IList<T>> matchedLinks = null;
148        //    if (matchedHandler != null)
149        //    {
150        //        matchedLinks = new List<IList<T>>();
151        //        Func<T, bool> handler = link =>
152        //        {
153        //            var matchedLink = Memory.GetLinkValue(link);
154        //            var matchDecision = matchedHandler(matchedLink);
155        //            if (Equals(matchDecision, Constants.Break))
156        //                return false;
157        //            if (!Equals(matchDecision, Constants.Skip))
158        //                matchedLinks.Add(matchedLink);
159        //            return true;
160        //        };
161        //        if (!Memory.Each(handler, restriction))
162        //            return Constants.Break;
163        //    }
164        //    if (!matchedLinks.IsNullOrEmpty())
165        //    {
166        //        var totalMatchedLinks = matchedLinks.Count;
167        //        for (var i = 0; i < totalMatchedLinks; i++)

```

```

167         //      {
168         //          var matchedLink = matchedLinks[i];
169         //          if (substitutedHandler != null)
170         //          {
171         //              var newValue = new List<T>(); // TODO: Prepare value to update here
172         //              // TODO: Decide is it actually needed to use Before and After
173         //              substitution handling.
174         //              var substitutedDecision = substitutedHandler(matchedLink,
175         //              newValue);
176         //              if (Equals(substitutedDecision, Constants.Break))
177         //                  return Constants.Break;
178         //              if (Equals(substitutedDecision, Constants.Continue))
179         //              {
180         //                  // Actual update here
181         //                  Memory.SetLinkValue(newValue);
182         //              }
183         //              if (Equals(substitutedDecision, Constants.Skip))
184         //              {
185         //                  // Cancel the update. TODO: decide use separate Cancel
186         //                  // constant or Skip is enough?
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         //  }
1000        //  }

```

```

if (patternOrCondition.Count == 1)
{
    var linkToDelete = patternOrCondition[0];
    var before = Links.GetLink(linkToDelete);
    if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
        ↪ Constants.Break))
    {
        return Constants.Break;
    }
    var after = ArrayPool<TLink>.Empty;
    Links.Update(linkToDelete, Constants.Null, Constants.Null);
    Links.Delete(linkToDelete);
    if (matchHandler != null)
    {
        return substitutionHandler(before, after);
    }
    return Constants.Continue;
}
else
{
    throw new NotSupportedException();
}
}
else // Replace / Update
{
    if (patternOrCondition.Count == 1) //-V3125
    {
        var linkToUpdate = patternOrCondition[0];
        var before = Links.GetLink(linkToUpdate);
        if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
            ↪ Constants.Break))
        {
            return Constants.Break;
        }
        var after = (IList<TLink>)substitution.ToArray(); //-V3125
        if (_equalityComparer.Equals(after[0], default))
        {
            after[0] = linkToUpdate;
        }
        if (substitution.Count == 1)
        {
            if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
            {
                after = Links.GetLink(substitution[0]);
                Links.Update(linkToUpdate, Constants.Null, Constants.Null);
                Links.Delete(linkToUpdate);
            }
        }
        else if (substitution.Count == 3)
        {
            Links.Update(after);
        }
        else
        {
            throw new NotSupportedException();
        }
        if (matchHandler != null)
        {
            return substitutionHandler(before, after);
        }
        return Constants.Continue;
    }
    else
    {
        throw new NotSupportedException();
    }
}
}
}

/// <remarks>
/// IList[IList[IList[T]]]
/// |         |         |         |
/// |         |         |-----|
/// |         |         |   link   |
/// |         |-----|
/// |         |   change   |
/// |-----|
/// |   changes   |
/// |

```

```

313     /// </remarks>
314     public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
        ↳ substitution)
315     {
316         var changes = new List<IList<IList<TLink>>>();
317         Trigger(condition, AlwaysContinue, substitution, (before, after) =>
318         {
319             var change = new[] { before, after };
320             changes.Add(change);
321             return Constants.Continue;
322         });
323         return changes;
324     }
325
326     private TLink AlwaysContinue(IList<TLink> linkToMatch) => Constants.Continue;
327 }
328 }

```

#### ./Platform.Data.Doublets/DoubletComparer.cs

```

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

```

#### ./Platform.Data.Doublets/Doublet.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets
5  {
6      public struct Doublet<T> : IEquatable<Doublet<T>>
7      {
8          private static readonly EqualityComparer<T> _equalityComparer =
            ↳ EqualityComparer<T>.Default;
9
10         public T Source { get; set; }
11         public T Target { get; set; }
12
13         public Doublet(T source, T target)
14         {
15             Source = source;
16             Target = target;
17         }
18
19         public override string ToString() => $"{Source}->{Target}";
20
21         public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
            ↳ && _equalityComparer.Equals(Target, other.Target);
22
23         public override bool Equals(object obj) => obj is Doublet<T> doublet ?
            ↳ base.Equals(doublet) : false;
24
25         public override int GetHashCode() => (Source, Target).GetHashCode();
26     }
27 }

```

#### ./Platform.Data.Doublets/Hybrid.cs

```

1  using System;
2  using System.Reflection;
3  using Platform.Reflection;
4  using Platform.Converters;
5  using Platform.Exceptions;
6

```

```

7 namespace Platform.Data.Doublets
8 {
9     public class Hybrid<T>
10    {
11        public readonly T Value;
12        public bool IsNothing => Convert.ToInt64(To.Signed(Value)) == 0;
13        public bool IsInternal => Convert.ToInt64(To.Signed(Value)) > 0;
14        public bool IsExternal => Convert.ToInt64(To.Signed(Value)) < 0;
15        public long AbsoluteValue => Numbers.Math.Abs(Convert.ToInt64(To.Signed(Value)));
16
17        public Hybrid(T value)
18        {
19            Ensure.Always.IsUnsignedInteger<T>();
20            Value = value;
21        }
22
23        public Hybrid(object value) => Value = To.UnsignedAs<T>(Convert.ChangeType(value,
24            ↪ Type<T>.SignedVersion));
25
26        public Hybrid(object value, bool isExternal)
27        {
28            var signedType = Type<T>.SignedVersion;
29            var signedValue = Convert.ChangeType(value, signedType);
30            var abs = typeof(Numbers.Math).GetTypeInfo().GetMethod("Abs").MakeGenericMethod(sign
31            ↪ edType);
32            var negate = typeof(Numbers.Math).GetTypeInfo().GetMethod("Negate").MakeGenericMetho
33            ↪ d(signedType);
34            var absoluteValue = abs.Invoke(null, new[] { signedValue });
35            var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
36            ↪ absoluteValue;
37            Value = To.UnsignedAs<T>(resultValue);
38        }
39
40        public static implicit operator Hybrid<T>(T integer) => new Hybrid<T>(integer);
41
42        public static explicit operator Hybrid<T>(ulong integer) => new Hybrid<T>(integer);
43
44        public static explicit operator Hybrid<T>(long integer) => new Hybrid<T>(integer);
45
46        public static explicit operator Hybrid<T>(uint integer) => new Hybrid<T>(integer);
47
48        public static explicit operator Hybrid<T>(int integer) => new Hybrid<T>(integer);
49
50        public static explicit operator Hybrid<T>(ushort integer) => new Hybrid<T>(integer);
51
52        public static explicit operator Hybrid<T>(short integer) => new Hybrid<T>(integer);
53
54        public static explicit operator Hybrid<T>(byte integer) => new Hybrid<T>(integer);
55
56        public static explicit operator Hybrid<T>(sbyte integer) => new Hybrid<T>(integer);
57
58        public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;
59
60        public static explicit operator ulong(Hybrid<T> hybrid) =>
61            ↪ Convert.ToUInt64(hybrid.Value);
62
63        public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;
64
65        public static explicit operator uint(Hybrid<T> hybrid) => Convert.ToUInt32(hybrid.Value);
66
67        public static explicit operator int(Hybrid<T> hybrid) =>
68            ↪ Convert.ToInt32(hybrid.AbsoluteValue);
69
70        public static explicit operator ushort(Hybrid<T> hybrid) =>
71            ↪ Convert.ToUInt16(hybrid.Value);
72
73        public static explicit operator short(Hybrid<T> hybrid) =>
74            ↪ Convert.ToInt16(hybrid.AbsoluteValue);
75
76        public static explicit operator byte(Hybrid<T> hybrid) => Convert.ToByte(hybrid.Value);
77
78        public static explicit operator sbyte(Hybrid<T> hybrid) =>
79            ↪ Convert.ToSByte(hybrid.AbsoluteValue);
80
81        public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
82            ↪ default(T).ToString() : IsExternal ? $"{AbsoluteValue}" : Value.ToString();
83    }
84 }

```

./Platform.Data.Doublets/ILinks.cs

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

./Platform.Data.Doublets/ILinksExtensions.cs

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



```

64     for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
        ↪ Arithmetic.Decrement(i))
65     {
66         links.Delete(i);
67         if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
68         {
69             i = links.Count();
70         }
71     }
72 }
73
74 public static TLink First<TLink>(this ILinks<TLink> links)
75 {
76     TLink firstLink = default;
77     var equalityComparer = EqualityComparer<TLink>.Default;
78     if (equalityComparer.Equals(links.Count(), default))
79     {
80         throw new Exception("В хранилище нет связей.");
81     }
82     links.Each(links.Constants.Any, links.Constants.Any, link =>
83     {
84         firstLink = link[links.Constants.IndexPart];
85         return links.Constants.Break;
86     });
87     if (equalityComparer.Equals(firstLink, default))
88     {
89         throw new Exception("В процессе поиска по хранилищу не было найдено связей.");
90     }
91     return firstLink;
92 }
93
94 public static bool IsInnerReference<TLink>(this ILinks<TLink> links, TLink reference)
95 {
96     var constants = links.Constants;
97     var comparer = Comparer<TLink>.Default;
98     return comparer.Compare(constants.MinPossibleIndex, reference) >= 0 &&
        ↪ comparer.Compare(reference, constants.MaxPossibleIndex) <= 0;
99 }
100
101 #region Paths
102
103 /// <remarks>
104 /// TODO: Как так? Как то что ниже может быть корректно?
105 /// Скорее всего практически не применимо
106 /// Предполагалось, что можно было конвертировать формируемый в проходе через
        ↪ SequenceWalker
107 /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
108 /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
109 /// </remarks>
110 public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
        ↪ path)
111 {
112     var current = path[0];
113     //EnsureLinkExists(current, "path");
114     if (!links.Exists(current))
115     {
116         return false;
117     }
118     var equalityComparer = EqualityComparer<TLink>.Default;
119     var constants = links.Constants;
120     for (var i = 1; i < path.Length; i++)
121     {
122         var next = path[i];
123         var values = links.GetLink(current);
124         var source = values[constants.SourcePart];
125         var target = values[constants.TargetPart];
126         if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
            ↪ next))
127         {
128             //throw new Exception(string.Format("Невозможно выбрать путь, так как и
            ↪ Source и Target совпадают с элементом пути {0}.", next));
129             return false;
130         }
131         if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
            ↪ target))
132         {
133             //throw new Exception(string.Format("Невозможно продолжить путь через
            ↪ элемент пути {0}", next));

```

```

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

```

```

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

```

```

256 [MethodImpl(MethodImplOptions.AggressiveInlining)]
257 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↪ Func<IList<TLink>, TLink> handler)
258 {
259     var constants = links.Constants;
260     return links.Each(handler, constants.Any, source, target);
261 }
262
263 [MethodImpl(MethodImplOptions.AggressiveInlining)]
264 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
    ↪ restrictions)
265 {
266     long arraySize = (Integer<TLink>)links.Count(restrictions);
267     var array = new IList<TLink>[arraySize];
268     if (arraySize > 0)
269     {
270         var filler = new ArrayFiller<IList<TLink>, TLink>(array,
            ↪ links.Constants.Continue);
271         links.Each(filler.AddAndReturnConstant, restrictions);
272     }
273     return array;
274 }
275
276 [MethodImpl(MethodImplOptions.AggressiveInlining)]
277 public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
    ↪ restrictions)
278 {
279     long arraySize = (Integer<TLink>)links.Count(restrictions);
280     var array = new TLink[arraySize];
281     if (arraySize > 0)
282     {
283         var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
284         links.Each(filler.AddFirstAndReturnConstant, restrictions);
285     }
286     return array;
287 }
288
289 /// <summary>
290 /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
    ↪ в хранилище связей.
291 /// </summary>
292 /// <param name="links">Хранилище связей.</param>
293 /// <param name="source">Начало связи.</param>
294 /// <param name="target">Конец связи.</param>
295 /// <returns>Значение, определяющее существует ли связь.</returns>
296 [MethodImpl(MethodImplOptions.AggressiveInlining)]
297 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;
298
299 #region Ensure
300 // TODO: May be move to EnsureExtensions or make it both there and here
301
302 [MethodImpl(MethodImplOptions.AggressiveInlining)]
303 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
    ↪ reference, string argumentName)
304 {
305     if (links.IsInnerReference(reference) && !links.Exists(reference))
306     {
307         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
308     }
309 }
310
311 [MethodImpl(MethodImplOptions.AggressiveInlining)]
312 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
    ↪ IList<TLink> restrictions, string argumentName)
313 {
314     for (int i = 0; i < restrictions.Count; i++)
315     {
316         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
317     }
318 }
319
320 [MethodImpl(MethodImplOptions.AggressiveInlining)]
321 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↪ restrictions)
322 {
323     for (int i = 0; i < restrictions.Count; i++)

```

```

324     {
325         links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
326     }
327 }
328
329 [MethodImpl(MethodImplOptions.AggressiveInlining)]
330 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
331     → string argumentName)
332 {
333     var equalityComparer = EqualityComparer<TLink>.Default;
334     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
335     {
336         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
337     }
338 }
339
340 [MethodImpl(MethodImplOptions.AggressiveInlining)]
341 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
342     → link, string argumentName)
343 {
344     var equalityComparer = EqualityComparer<TLink>.Default;
345     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
346     {
347         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
348     }
349 }
350
351 /// <param name="links">Хранилище связей.</param>
352 [MethodImpl(MethodImplOptions.AggressiveInlining)]
353 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
354     → TLink target)
355 {
356     if (links.Exists(source, target))
357     {
358         throw new LinkWithSameValueAlreadyExistsException();
359     }
360 }
361
362 /// <param name="links">Хранилище связей.</param>
363 public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
364 {
365     if (links.HasUsages(link))
366     {
367         throw new ArgumentLinkHasDependenciesException<TLink>(link);
368     }
369 }
370
371 /// <param name="links">Хранилище связей.</param>
372 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
373     → addresses) => links.EnsureCreated(links.Create, addresses);
374
375 /// <param name="links">Хранилище связей.</param>
376 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
377     → addresses) => links.EnsureCreated(links.CreatePoint, addresses);
378
379 /// <param name="links">Хранилище связей.</param>
380 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
381     → params TLink[] addresses)
382 {
383     var constants = links.Constants;
384     var nonExistentAddresses = new HashSet<ulong>(addresses.Where(x =>
385     → !links.Exists(x)).Select(x => (ulong)(Integer<TLink>)x));
386     if (nonExistentAddresses.Count > 0)
387     {
388         var max = nonExistentAddresses.Max();
389         // TODO: Эту верхнюю границу нужно разрешить переопределять (проверить
390         → применяется ли эта логика)
391         max = System.Math.Min(max, (Integer<TLink>)constants.MaxPossibleIndex);
392         var createdLinks = new List<TLink>();
393         var equalityComparer = EqualityComparer<TLink>.Default;
394         TLink createdLink = creator();
395         while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
396         {
397             createdLinks.Add(createdLink);
398         }
399         for (var i = 0; i < createdLinks.Count; i++)
400         {
401             if (!nonExistentAddresses.Contains((Integer<TLink>)createdLinks[i]))

```

```

394         {
395             links.Delete(createdLinks[i]);
396         }
397     }
398 }
399 }
400
401 #endregion
402
403 /// <param name="links">Хранилище связей.</param>
404 public static ulong CountUsages<TLink>(this ILinks<TLink> links, TLink link)
405 {
406     var constants = links.Constants;
407     var values = links.GetLink(link);
408     ulong usagesAsSource = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
409         ↪ link, constants.Any));
410     var equalityComparer = EqualityComparer<TLink>.Default;
411     if (equalityComparer.Equals(values[constants.SourcePart], link))
412     {
413         usagesAsSource--;
414     }
415     ulong usagesAsTarget = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
416         ↪ constants.Any, link));
417     if (equalityComparer.Equals(values[constants.TargetPart], link))
418     {
419         usagesAsTarget--;
420     }
421     return usagesAsSource + usagesAsTarget;
422 }
423
424 /// <param name="links">Хранилище связей.</param>
425 [MethodImpl(MethodImplOptions.AggressiveInlining)]
426 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
427     ↪ links.CountUsages(link) > 0;
428
429 /// <param name="links">Хранилище связей.</param>
430 [MethodImpl(MethodImplOptions.AggressiveInlining)]
431 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
432     ↪ TLink target)
433 {
434     var constants = links.Constants;
435     var values = links.GetLink(link);
436     var equalityComparer = EqualityComparer<TLink>.Default;
437     return equalityComparer.Equals(values[constants.SourcePart], source) &&
438         ↪ equalityComparer.Equals(values[constants.TargetPart], target);
439 }
440
441 /// <summary>
442 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
443 /// </summary>
444 /// <param name="links">Хранилище связей.</param>
445 /// <param name="source">Индекс связи, которая является началом для искомой
446     ↪ связи.</param>
447 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
448 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
449     ↪ (концом).</returns>
450 [MethodImpl(MethodImplOptions.AggressiveInlining)]
451 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
452     ↪ target)
453 {
454     var constants = links.Constants;
455     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
456     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
457     return setter.Result;
458 }
459
460 /// <param name="links">Хранилище связей.</param>
461 [MethodImpl(MethodImplOptions.AggressiveInlining)]
462 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
463 {
464     var link = links.Create();
465     return links.Update(link, link, link);
466 }
467
468 /// <param name="links">Хранилище связей.</param>
469 [MethodImpl(MethodImplOptions.AggressiveInlining)]
470 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
471     ↪ target) => links.Update(links.Create(), source, target);

```

```

463 /// <summary>
464 /// Обновляет связь с указанными началом (Source) и концом (Target)
465 /// на связь с указанными началом (NewSource) и концом (NewTarget).
466 /// </summary>
467 /// <param name="links">Хранилище связей.</param>
468 /// <param name="link">Индекс обновляемой связи.</param>
469 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
470   ↳ выполняется обновление.</param>
471 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
472   ↳ выполняется обновление.</param>
473 /// <returns>Индекс обновлённой связи.</returns>
474 [MethodImpl(MethodImplOptions.AggressiveInlining)]
475 public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
476   ↳ TLink newTarget) => links.Update(new Link<TLink>(link, newSource, newTarget));
477
478 /// <summary>
479 /// Обновляет связь с указанными началом (Source) и концом (Target)
480 /// на связь с указанными началом (NewSource) и концом (NewTarget).
481 /// </summary>
482 /// <param name="links">Хранилище связей.</param>
483 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
484   ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
485   ↳ Itself - требование установить ссылку на себя, 1.. $\infty$  конкретный адрес другой
486   ↳ связи.</param>
487 /// <returns>Индекс обновлённой связи.</returns>
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
490 {
491     if (restrictions.Length == 2)
492     {
493         return links.MergeAndDelete(restrictions[0], restrictions[1]);
494     }
495     if (restrictions.Length == 4)
496     {
497         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
498           ↳ restrictions[2], restrictions[3]);
499     }
500     else
501     {
502         return links.Update(restrictions);
503     }
504 }
505
506 [MethodImpl(MethodImplOptions.AggressiveInlining)]
507 public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
508   ↳ links, TLink constant, IList<TLink> restrictions)
509 {
510     var equalityComparer = EqualityComparer<TLink>.Default;
511     var constants = links.Constants;
512     var index = restrictions[constants.IndexPart];
513     var source = restrictions[constants.SourcePart];
514     var target = restrictions[constants.TargetPart];
515     source = equalityComparer.Equals(source, constant) ? index : source;
516     target = equalityComparer.Equals(target, constant) ? index : target;
517     return new Link<TLink>(index, source, target);
518 }
519
520 /// <summary>
521 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
522   ↳ с указанными Source (началом) и Target (концом).
523 /// </summary>
524 /// <param name="links">Хранилище связей.</param>
525 /// <param name="source">Индекс связи, которая является началом на создаваемой
526   ↳ связи.</param>
527 /// <param name="target">Индекс связи, которая является концом для создаваемой
528   ↳ связи.</param>
529 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
530 [MethodImpl(MethodImplOptions.AggressiveInlining)]
531 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
532   ↳ target)
533 {
534     var link = links.SearchOrDefault(source, target);
535     if (EqualityComparer<TLink>.Default.Equals(link, default))
536     {
537         link = links.CreateAndUpdate(source, target);
538     }
539 }

```

```

528     return link;
529 }
530
531 /// <summary>
532 /// Обновляет связь с указанными началом (Source) и концом (Target)
533 /// на связь с указанными началом (NewSource) и концом (NewTarget).
534 /// </summary>
535 /// <param name="links">Хранилище связей.</param>
536 /// <param name="source">Индекс связи, которая является началом обновляемой
    → связи.</param>
537 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
538 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    → выполняется обновление.</param>
539 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    → выполняется обновление.</param>
540 /// <returns>Индекс обновлённой связи.</returns>
541 [MethodImpl(MethodImplOptions.AggressiveInlining)]
542 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
    → TLink target, TLink newSource, TLink newTarget)
543 {
544     var equalityComparer = EqualityComparer<TLink>.Default;
545     var link = links.SearchOrDefault(source, target);
546     if (equalityComparer.Equals(link, default))
547     {
548         return links.CreateAndUpdate(newSource, newTarget);
549     }
550     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
    → target))
551     {
552         return link;
553     }
554     return links.Update(link, newSource, newTarget);
555 }
556
557 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
558 /// <param name="links">Хранилище связей.</param>
559 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
560 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
561 [MethodImpl(MethodImplOptions.AggressiveInlining)]
562 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
    → target)
563 {
564     var link = links.SearchOrDefault(source, target);
565     if (!EqualityComparer<TLink>.Default.Equals(link, default))
566     {
567         links.Delete(link);
568         return link;
569     }
570     return default;
571 }
572
573 /// <summary>Удаляет несколько связей.</summary>
574 /// <param name="links">Хранилище связей.</param>
575 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
576 [MethodImpl(MethodImplOptions.AggressiveInlining)]
577 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
578 {
579     for (int i = 0; i < deletedLinks.Count; i++)
580     {
581         links.Delete(deletedLinks[i]);
582     }
583 }
584
585 /// <remarks>Before execution of this method ensure that deleted link is detached (all
    → values - source and target are reset to null) or it might enter into infinite
    → recursion.</remarks>
586 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
587 {
588     var anyConstant = links.Constants.Any;
589     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
590     links.DeleteByQuery(usagesAsSourceQuery);
591     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
592     links.DeleteByQuery(usagesAsTargetQuery);
593 }
594
595 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
596 {
597     var count = (Integer<TLink>)links.Count(query);

```



```

598     if (count > 0)
599     {
600         var queryResult = new TLink[count];
601         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
        ↪ links.Constants.Continue);
602         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
603         for (var i = (long)count - 1; i >= 0; i--)
604         {
605             links.Delete(queryResult[i]);
606         }
607     }
608 }
609
610 // TODO: Move to Platform.Data
611 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
612 {
613     var nullConstant = links.Constants.Null;
614     var equalityComparer = EqualityComparer<TLink>.Default;
615     var link = links.GetLink(linkIndex);
616     for (int i = 1; i < link.Count; i++)
617     {
618         if (!equalityComparer.Equals(link[i], nullConstant))
619         {
620             return false;
621         }
622     }
623     return true;
624 }
625
626 // TODO: Create a universal version of this method in Platform.Data (with using of for
        ↪ loop)
627 public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
628 {
629     var nullConstant = links.Constants.Null;
630     var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
631     links.Update(updateRequest);
632 }
633
634 // TODO: Create a universal version of this method in Platform.Data (with using of for
        ↪ loop)
635 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
636 {
637     if (!links.AreValuesReset(linkIndex))
638     {
639         links.ResetValues(linkIndex);
640     }
641 }
642
643 /// <summary>
644 /// Merging two usages graphs, all children of old link moved to be children of new link
        ↪ or deleted.
645 /// </summary>
646 public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
        ↪ TLink newLinkIndex)
647 {
648     var equalityComparer = EqualityComparer<TLink>.Default;
649     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
650     {
651         var constants = links.Constants;
652         var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
        ↪ constants.Any);
653         long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
654         var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
        ↪ oldLinkIndex);
655         long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
656         var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
        ↪ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
657         if (!isStandalonePoint)
658         {
659             var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
660             if (totalUsages > 0)
661             {
662                 var usages = ArrayPool.Allocate<TLink>(totalUsages);
663                 var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
        ↪ links.Constants.Continue);
664                 var i = 0L;
665                 if (usagesAsSourceCount > 0)
666                 {

```

```

667         links.Each(usagesFiller.AddFirstAndReturnConstant,
        ↪ usagesAsSourceQuery);
668     for (; i < usagesAsSourceCount; i++)
669     {
670         var usage = usages[i];
671         if (!equalityComparer.Equals(usage, oldLinkIndex))
672         {
673             links.Update(usage, newLinkIndex, links.GetTarget(usage));
674         }
675     }
676 }
677 if (usagesAsTargetCount > 0)
678 {
679     links.Each(usagesFiller.AddFirstAndReturnConstant,
        ↪ usagesAsTargetQuery);
680     for (; i < usages.Length; i++)
681     {
682         var usage = usages[i];
683         if (!equalityComparer.Equals(usage, oldLinkIndex))
684         {
685             links.Update(usage, links.GetSource(usage), newLinkIndex);
686         }
687     }
688 }
689 ArrayPool.Free(usages);
690 }
691 }
692 }
693 return newLinkIndex;
694 }
695
696 /// <summary>
697 /// Replace one link with another (replaced link is deleted, children are updated or
        ↪ deleted).
698 /// </summary>
699 [MethodImpl(MethodImplOptions.AggressiveInlining)]
700 public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
        ↪ TLink newLinkIndex)
701 {
702     var equalityComparer = EqualityComparer<TLink>.Default;
703     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
704     {
705         links.MergeUsages(oldLinkIndex, newLinkIndex);
706         links.Delete(oldLinkIndex);
707     }
708     return newLinkIndex;
709 }
710 }
711 }

```

./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs

```

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

```

```

27     }
28     var source = Links.GetSource(frequency);
29     var incrementedSource = _unaryNumberIncrementer.Increment(source);
30     return Links.GetOrCreate(incrementedSource, _frequencyMarker);
31 }
32 }
33 }

```

#### ./Platform.Data.Doublets/Incrementers/LinkFrequencyIncrementer.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Incrementers
5  {
6      public class LinkFrequencyIncrementer<TLink> : LinksOperatorBase<TLink>,
7          ↳ IIncrementer<IList<TLink>>
8      {
9          private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
10         private readonly IIncrementer<TLink> _frequencyIncrementer;
11
12         public LinkFrequencyIncrementer(ILinks<TLink> links, IPropertyOperator<TLink, TLink>
13             ↳ frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
14             : base(links)
15         {
16             _frequencyPropertyOperator = frequencyPropertyOperator;
17             _frequencyIncrementer = frequencyIncrementer;
18
19             /// <remarks>Sequence itseft is not changed, only frequency of its doublets is
20             ↳ incremented.</remarks>
21         public IList<TLink> Increment(IList<TLink> sequence) // TODO: May be move to
22             ↳ ILinksExtensions or make SequenceDoubletsFrequencyIncrementer
23         {
24             for (var i = 1; i < sequence.Count; i++)
25             {
26                 Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
27             }
28             return sequence;
29         }
30
31         public void Increment(TLink link)
32         {
33             var previousFrequency = _frequencyPropertyOperator.Get(link);
34             var frequency = _frequencyIncrementer.Increment(previousFrequency);
35             _frequencyPropertyOperator.Set(link, frequency);
36         }
37     }
38 }

```

#### ./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs

```

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

```

```

28         return Links.GetOrCreate(source, Increment(target));
29     }
30 }
31 }
32 }

```

./Platform.Data.Doublets/ISynchronizedLinks.cs

```

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

```

./Platform.Data.Doublets/Link.cs

```

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

```

```

55 public static Link<TLink> Create(TLink source, TLink target) => new Link<TLink>(source,
56     ↪ target);
57
58 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
59
60 public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
61     && _equalityComparer.Equals(Source, _constants.Null)
62     && _equalityComparer.Equals(Target, _constants.Null);
63
64 public override bool Equals(object other) => other is Link<TLink> &&
65     ↪ Equals((Link<TLink>)other);
66
67 public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
68     && _equalityComparer.Equals(Source, other.Source)
69     && _equalityComparer.Equals(Target, other.Target);
70
71 public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
72     ↪ {source}->{target})";
73
74 public static string ToString(TLink source, TLink target) => $"({source}->{target})";
75
76 public static implicit operator TLink[](Link<TLink> link) => link.ToArray();
77
78 public static implicit operator Link<TLink>(TLink[] linkArray) => new
79     ↪ Link<TLink>(linkArray);
80
81 public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
82     ↪ ToString(Source, Target) : ToString(Index, Source, Target);
83
84 #region IList
85
86 public int Count => Length;
87
88 public bool IsReadOnly => true;
89
90 public TLink this[int index]
91 {
92     get
93     {
94         Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
95             ↪ nameof(index));
96         if (index == _constants.IndexPart)
97         {
98             return Index;
99         }
100         if (index == _constants.SourcePart)
101         {
102             return Source;
103         }
104         if (index == _constants.TargetPart)
105         {
106             return Target;
107         }
108         throw new NotSupportedException(); // Impossible path due to
109             ↪ Ensure.ArgumentInRange
110     }
111     set => throw new NotSupportedException();
112 }
113
114 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
115
116 public IEnumerator<TLink> GetEnumerator()
117 {
118     yield return Index;
119     yield return Source;
120     yield return Target;
121 }
122
123 public void Add(TLink item) => throw new NotSupportedException();
124
125 public void Clear() => throw new NotSupportedException();
126
127 public bool Contains(TLink item) => IndexOf(item) >= 0;
128
129 public void CopyTo(TLink[] array, int arrayIndex)
130 {
131     Ensure.Always.ArgumentNotNull(array, nameof(array));
132     Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
133         ↪ nameof(arrayIndex));

```

```

127         if (arrayIndex + Length > array.Length)
128         {
129             throw new InvalidOperationException();
130         }
131         array[arrayIndex++] = Index;
132         array[arrayIndex++] = Source;
133         array[arrayIndex] = Target;
134     }
135
136     public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
137
138     public int IndexOf(TLink item)
139     {
140         if (_equalityComparer.Equals(Index, item))
141         {
142             return _constants.IndexPart;
143         }
144         if (_equalityComparer.Equals(Source, item))
145         {
146             return _constants.SourcePart;
147         }
148         if (_equalityComparer.Equals(Target, item))
149         {
150             return _constants.TargetPart;
151         }
152         return -1;
153     }
154
155     public void Insert(int index, TLink item) => throw new NotSupportedException();
156
157     public void RemoveAt(int index) => throw new NotSupportedException();
158
159     #endregion
160 }
161 }

```

./Platform.Data.Doublets/LinkExtensions.cs

```

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

```

./Platform.Data.Doublets/LinksOperatorBase.cs

```

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

```

./Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs

```

1 using System.Linq;
2 using System.Collections.Generic;
3 using Platform.Interfaces;
4
5 namespace Platform.Data.Doublets.PropertyOperators
6 {
7     public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>,
8         ↪ IPropertiesOperator<TLink, TLink, TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
14
15         public TLink GetValue(TLink @object, TLink property)
16         {
17             var objectProperty = Links.SearchOrDefault(@object, property);
18             if (_equalityComparer.Equals(objectProperty, default))
19             {
20                 return default;
21             }
22         }
23     }
24 }

```

```

19     }
20     var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
21     if (valueLink == null)
22     {
23         return default;
24     }
25     return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
26 }
27
28 public void SetValue(TLink @object, TLink property, TLink value)
29 {
30     var objectProperty = Links.GetOrCreate(@object, property);
31     Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
32     Links.GetOrCreate(objectProperty, value);
33 }
34 }
35 }

```

./Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.PropertyOperators
5 {
6     public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IPropertyOperator<TLink,
7     ↪ TLink>
8     {
9         private static readonly EqualityComparer<TLink> _equalityComparer =
10         ↪ EqualityComparer<TLink>.Default;
11
12         private readonly TLink _propertyMarker;
13         private readonly TLink _propertyValueMarker;
14
15         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
16         ↪ propertyValueMarker) : base(links)
17         {
18             _propertyMarker = propertyMarker;
19             _propertyValueMarker = propertyValueMarker;
20         }
21
22         public TLink Get(TLink link)
23         {
24             var property = Links.SearchOrDefault(link, _propertyMarker);
25             var container = GetContainer(property);
26             var value = GetValue(container);
27             return value;
28         }
29
30         private TLink GetContainer(TLink property)
31         {
32             var valueContainer = default(TLink);
33             if (_equalityComparer.Equals(property, default))
34             {
35                 return valueContainer;
36             }
37             var constants = Links.Constants;
38             var countinueConstant = constants.Continue;
39             var breakConstant = constants.Break;
40             var anyConstant = constants.Any;
41             var query = new Link<TLink>(anyConstant, property, anyConstant);
42             Links.Each(candidate =>
43             {
44                 var candidateTarget = Links.GetTarget(candidate);
45                 var valueTarget = Links.GetTarget(candidateTarget);
46                 if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
47                 {
48                     valueContainer = Links.GetIndex(candidate);
49                     return breakConstant;
50                 }
51                 return countinueConstant;
52             }, query);
53             return valueContainer;
54         }
55
56         private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
57         ↪ ? default : Links.GetTarget(container);
58
59         public void Set(TLink link, TLink value)
60         {
61             var property = Links.GetOrCreate(link, _propertyMarker);

```

```

58         var container = GetContainer(property);
59         if (_equalityComparer.Equals(container, default))
60         {
61             Links.GetOrCreate(property, value);
62         }
63         else
64         {
65             Links.Update(container, property, value);
66         }
67     }
68 }
69 }

```

./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.cs

```

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

```



```

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

```

```

113     public static TLink GetAllocatedLinks(IntPtr pointer) => (pointer +
114         ↪ AllocatedLinksOffset).GetValue<TLink>();
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     public static TLink GetReservedLinks(IntPtr pointer) => (pointer +
117         ↪ ReservedLinksOffset).GetValue<TLink>();
118     [MethodImpl(MethodImplOptions.AggressiveInlining)]
119     public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
120         ↪ FreeLinksOffset).GetValue<TLink>();
121     [MethodImpl(MethodImplOptions.AggressiveInlining)]
122     public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
123         ↪ FirstFreeLinkOffset).GetValue<TLink>();
124     [MethodImpl(MethodImplOptions.AggressiveInlining)]
125     public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
126         ↪ FirstAsSourceOffset).GetValue<TLink>();
127     [MethodImpl(MethodImplOptions.AggressiveInlining)]
128     public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
129         ↪ FirstAsTargetOffset).GetValue<TLink>();
130     [MethodImpl(MethodImplOptions.AggressiveInlining)]
131     public static TLink GetLastFreeLink(IntPtr pointer) => (pointer +
132         ↪ LastFreeLinkOffset).GetValue<TLink>();
133
134     [MethodImpl(MethodImplOptions.AggressiveInlining)]
135     public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) => pointer +
136         ↪ FirstAsSourceOffset;
137     [MethodImpl(MethodImplOptions.AggressiveInlining)]
138     public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
139         ↪ FirstAsTargetOffset;
140
141     [MethodImpl(MethodImplOptions.AggressiveInlining)]
142     public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
143         ↪ AllocatedLinksOffset).SetValue(value);
144     [MethodImpl(MethodImplOptions.AggressiveInlining)]
145     public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
146         ↪ ReservedLinksOffset).SetValue(value);
147     [MethodImpl(MethodImplOptions.AggressiveInlining)]
148     public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
149         ↪ FreeLinksOffset).SetValue(value);
150     [MethodImpl(MethodImplOptions.AggressiveInlining)]
151     public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
152         ↪ FirstFreeLinkOffset).SetValue(value);
153     [MethodImpl(MethodImplOptions.AggressiveInlining)]
154     public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
155         ↪ FirstAsSourceOffset).SetValue(value);
156     [MethodImpl(MethodImplOptions.AggressiveInlining)]
157     public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
158         ↪ FirstAsTargetOffset).SetValue(value);
159     [MethodImpl(MethodImplOptions.AggressiveInlining)]
160     public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
161         ↪ LastFreeLinkOffset).SetValue(value);
162 }
163
164 private readonly long _memoryReservationStep;
165
166 private readonly IResizableDirectMemory _memory;
167 private IntPtr _header;
168 private IntPtr _links;
169
170 private LinksTargetsTreeMethods _targetsTreeMethods;
171 private LinksSourcesTreeMethods _sourcesTreeMethods;
172
173 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
174 ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
175 ↪ наличие связи внутри
176 private UnusedLinksListMethods _unusedLinksListMethods;
177
178 /// <summary>
179 /// Возвращает общее число связей находящихся в хранилище.
180 /// </summary>
181 private TLink Total => Subtract(LinksHeader.GetAllocatedLinks(_header),
182     ↪ LinksHeader.GetFreeLinks(_header));
183
184 public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
185
186 public ResizableDirectMemoryLinks(string address)
187     : this(address, DefaultLinksSizeStep)
188 {
189 }

```

```

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

```

```

243         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
244             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
245         {
246             return Integer<TLink>.One;
247         }
248         return Integer<TLink>.Zero;
249     }
250 }
251 if (restrictions.Count == 3)
252 {
253     var index = restrictions[Constants.IndexPart];
254     var source = restrictions[Constants.SourcePart];
255     var target = restrictions[Constants.TargetPart];
256
257     if (_equalityComparer.Equals(index, Constants.Any))
258     {
259         if (_equalityComparer.Equals(source, Constants.Any) &&
260             ↪ _equalityComparer.Equals(target, Constants.Any))
261         {
262             return Total;
263         }
264         else if (_equalityComparer.Equals(source, Constants.Any))
265         {
266             return _targetsTreeMethods.CountUsages(target);
267         }
268         else if (_equalityComparer.Equals(target, Constants.Any))
269         {
270             return _sourcesTreeMethods.CountUsages(source);
271         }
272         else //if(source != Any && target != Any)
273         {
274             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
275             var link = _sourcesTreeMethods.Search(source, target);
276             return _equalityComparer.Equals(link, Constants.Null) ?
277                 ↪ Integer<TLink>.Zero : Integer<TLink>.One;
278         }
279     }
280     else
281     {
282         if (!Exists(index))
283         {
284             return Integer<TLink>.Zero;
285         }
286         if (_equalityComparer.Equals(source, Constants.Any) &&
287             ↪ _equalityComparer.Equals(target, Constants.Any))
288         {
289             return Integer<TLink>.One;
290         }
291         var storedLinkValue = GetLinkUnsafe(index);
292         if (!_equalityComparer.Equals(source, Constants.Any) &&
293             ↪ !_equalityComparer.Equals(target, Constants.Any))
294         {
295             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
296                 _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
297             {
298                 return Integer<TLink>.One;
299             }
300             return Integer<TLink>.Zero;
301         }
302         var value = default(TLink);
303         if (_equalityComparer.Equals(source, Constants.Any))
304         {
305             value = target;
306         }
307         if (_equalityComparer.Equals(target, Constants.Any))
308         {
309             value = source;
310         }
311         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
312             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
313         {
314             return Integer<TLink>.One;
315         }
316         return Integer<TLink>.Zero;
317     }
318 }
319 throw new NotSupportedException("Другие размеры и способы ограничений не
320     ↪ поддерживаются.");

```

```

316 }
317
318 [MethodImpl(MethodImplOptions.AggressiveInlining)]
319 public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
320 {
321     if (restrictions.Count == 0)
322     {
323         for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
324             ↪ (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
325             ↪ Increment(link))
326         {
327             if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
328                 ↪ Constants.Break))
329             {
330                 return Constants.Break;
331             }
332         }
333         return Constants.Continue;
334     }
335     if (restrictions.Count == 1)
336     {
337         var index = restrictions[Constants.IndexPart];
338         if (_equalityComparer.Equals(index, Constants.Any))
339         {
340             return Each(handler, ArrayPool<TLink>.Empty);
341         }
342         if (!Exists(index))
343         {
344             return Constants.Continue;
345         }
346         return handler(GetLinkStruct(index));
347     }
348     if (restrictions.Count == 2)
349     {
350         var index = restrictions[Constants.IndexPart];
351         var value = restrictions[1];
352         if (_equalityComparer.Equals(index, Constants.Any))
353         {
354             if (_equalityComparer.Equals(value, Constants.Any))
355             {
356                 return Each(handler, ArrayPool<TLink>.Empty);
357             }
358             if (_equalityComparer.Equals(Each(handler, new[] { index, value,
359                 ↪ Constants.Any })), Constants.Break))
360             {
361                 return Constants.Break;
362             }
363             return Each(handler, new[] { index, Constants.Any, value });
364         }
365         else
366         {
367             if (!Exists(index))
368             {
369                 return Constants.Continue;
370             }
371             if (_equalityComparer.Equals(value, Constants.Any))
372             {
373                 return handler(GetLinkStruct(index));
374             }
375             var storedLinkValue = GetLinkUnsafe(index);
376             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
377                 ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
378             {
379                 return handler(GetLinkStruct(index));
380             }
381             return Constants.Continue;
382         }
383     }
384     if (restrictions.Count == 3)
385     {
386         var index = restrictions[Constants.IndexPart];
387         var source = restrictions[Constants.SourcePart];
388         var target = restrictions[Constants.TargetPart];
389         if (_equalityComparer.Equals(index, Constants.Any))
390         {
391             if (_equalityComparer.Equals(source, Constants.Any) &&
392                 ↪ _equalityComparer.Equals(target, Constants.Any))

```

```

389         {
390             return Each(handler, ArrayPool<TLink>.Empty);
391         }
392     else if (_equalityComparer.Equals(source, Constants.Any))
393     {
394         return _targetsTreeMethods.EachUsage(target, handler);
395     }
396     else if (_equalityComparer.Equals(target, Constants.Any))
397     {
398         return _sourcesTreeMethods.EachUsage(source, handler);
399     }
400     else //if(source != Any && target != Any)
401     {
402         var link = _sourcesTreeMethods.Search(source, target);
403         return _equalityComparer.Equals(link, Constants.Null) ?
            ↪ Constants.Continue : handler(GetLinkStruct(link));
404     }
405 }
406 else
407 {
408     if (!Exists(index))
409     {
410         return Constants.Continue;
411     }
412     if (_equalityComparer.Equals(source, Constants.Any) &&
        ↪ _equalityComparer.Equals(target, Constants.Any))
413     {
414         return handler(GetLinkStruct(index));
415     }
416     var storedLinkValue = GetLinkUnsafe(index);
417     if (!_equalityComparer.Equals(source, Constants.Any) &&
        ↪ !_equalityComparer.Equals(target, Constants.Any))
418     {
419         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
            ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
420         {
421             return handler(GetLinkStruct(index));
422         }
423         return Constants.Continue;
424     }
425     var value = default(TLink);
426     if (_equalityComparer.Equals(source, Constants.Any))
427     {
428         value = target;
429     }
430     if (_equalityComparer.Equals(target, Constants.Any))
431     {
432         value = source;
433     }
434     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
        ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
435     {
436         return handler(GetLinkStruct(index));
437     }
438     return Constants.Continue;
439 }
440 }
441 }
442 }
443 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↪ поддерживаются.");
444 }
445
446 /// <remarks>
447 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↪ в другом месте (но не в менеджере памяти, а в логике Links)
448 /// </remarks>
449 [MethodImpl(MethodImplOptions.AggressiveInlining)]
450 public TLink Update(ICollection<TLink> values)
451 {
452     var linkIndex = values[Constants.IndexPart];
453     var link = GetLinkUnsafe(linkIndex);
454     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↪ предварительно заполнено нулями
455     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
456     {
457         _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
            ↪ linkIndex);
458     }
459     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))

```

```

460     {
461         _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
462             ↪ linkIndex);
463     }
464     Link.SetSource(link, values[Constants.SourcePart]);
465     Link.SetTarget(link, values[Constants.TargetPart]);
466     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
467     {
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) => _links.GetElement(LinkSizeInBytes,
488     ↪ linkIndex);
489
490 /// <remarks>
491 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
492 ↪ пространство
493 /// </remarks>
494 public TLink Create()
495 {
496     var freeLink = LinksHeader.GetFirstFreeLink(_header);
497     if (!_equalityComparer.Equals(freeLink, Constants.Null))
498     {
499         _unusedLinksListMethods.Detach(freeLink);
500     }
501     else
502     {
503         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
504             ↪ Constants.MaxPossibleIndex) > 0)
505         {
506             throw new
507                 ↪ LinksLimitReachedException((Integer<TLink>)Constants.MaxPossibleIndex);
508         }
509         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
510             ↪ Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
511         {
512             _memory.ReservedCapacity += _memory.ReservationStep;
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         _unusedLinksListMethods.AttachAsFirst(link);
530     }
531     else if (!_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
532     {
533         LinksHeader.SetAllocatedLinks(_header,
534             ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
535         _memory.UsedCapacity -= LinkSizeInBytes;
536         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
537         ↪ пока не дойдём до первой существующей связи

```

```

526 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
527 while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
    ↳ Integer<TLink>.Zero) > 0) &&
    ↳ IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
528 {
529     _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
530     LinksHeader.SetAllocatedLinks(_header,
    ↳ Decrement(LinksHeader.GetAllocatedLinks(_header)));
531     _memory.UsedCapacity -= LinkSizeInBytes;
532 }
533 }
534 }
535
536 /// <remarks>
537 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
    ↳ адрес реально поменялся
538 ///
539 /// Указатель this.links может быть в том же месте,
540 /// так как 0-я связь не используется и имеет такой же размер как Header,
541 /// поэтому header размещается в том же месте, что и 0-я связь
542 /// </remarks>
543 private void SetPointers(IDirectMemory memory)
544 {
545     if (memory == null)
546     {
547         _links = IntPtr.Zero;
548         _header = _links;
549         _unusedLinksListMethods = null;
550         _targetsTreeMethods = null;
551         _unusedLinksListMethods = null;
552     }
553     else
554     {
555         _links = memory.Pointer;
556         _header = _links;
557         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
558         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
559         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
560     }
561 }
562
563 [MethodImpl(MethodImplOptions.AggressiveInlining)]
564 private bool Exists(TLink link)
565 => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
566     && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
567     && !IsUnusedLink(link);
568
569 [MethodImpl(MethodImplOptions.AggressiveInlining)]
570 private bool IsUnusedLink(TLink link)
571 => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
572     || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)),
    ↳ Constants.Null)
    ↳ && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)), Constants.Null));
573
574 #region DisposableBase
575
576 protected override bool AllowMultipleDisposeCalls => true;
577
578 protected override void Dispose(bool manual, bool wasDisposed)
579 {
580     if (!wasDisposed)
581     {
582         SetPointers(null);
583         _memory.DisposeIfPossible();
584     }
585 }
586
587 #endregion
588
589 }
590 }

```

./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs

```

1 using System;
2 using Platform.Unsafe;
3 using Platform.Collections.Methods.Lists;
4
5 namespace Platform.Data.Doublets.ResizableDirectMemory
6 {
7     partial class ResizableDirectMemoryLinks<TLink>
8     {

```



```

9     private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<TLink>
10    {
11        private readonly IntPtr _links;
12        private readonly IntPtr _header;
13
14        public UnusedLinksListMethods(IntPtr links, IntPtr header)
15        {
16            _links = links;
17            _header = header;
18        }
19
20        protected override TLink GetFirst() => (_header +
21        ↪ LinksHeader.FirstFreeLinkOffset).GetValue<TLink>();
22
23        protected override TLink GetLast() => (_header +
24        ↪ LinksHeader.LastFreeLinkOffset).GetValue<TLink>();
25
26        protected override TLink GetPrevious(TLink element) =>
27        ↪ (_links.GetElement(LinkSizeInBytes, element) +
28        ↪ Link.SourceOffset).GetValue<TLink>();
29
30        protected override TLink GetNext(TLink element) =>
31        ↪ (_links.GetElement(LinkSizeInBytes, element) +
32        ↪ Link.TargetOffset).GetValue<TLink>();
33
34        protected override TLink GetSize() => (_header +
35        ↪ LinksHeader.FreeLinksOffset).GetValue<TLink>();
36
37        protected override void SetFirst(TLink element) => (_header +
38        ↪ LinksHeader.FirstFreeLinkOffset).SetValue(element);
39
40        protected override void SetLast(TLink element) => (_header +
41        ↪ LinksHeader.LastFreeLinkOffset).SetValue(element);
42
43        protected override void SetPrevious(TLink element, TLink previous) =>
44        ↪ (_links.GetElement(LinkSizeInBytes, element) +
45        ↪ Link.SourceOffset).SetValue(previous);
46
47        protected override void SetNext(TLink element, TLink next) =>
48        ↪ (_links.GetElement(LinkSizeInBytes, element) + Link.TargetOffset).SetValue(next);
49
50        protected override void SetSize(TLink size) => (_header +
51        ↪ LinksHeader.FreeLinksOffset).SetValue(size);
52    }
53 }

```

./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs

```

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

```

```

32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 protected abstract TLink GetBasePartValue(TLink link);
34
35 public TLink this[TLink index]
36 {
37     get
38     {
39         var root = GetTreeRoot();
40         if (GreaterOrEqualThan(index, GetSize(root)))
41         {
42             return GetZero();
43         }
44         while (!EqualToZero(root))
45         {
46             var left = GetLeftOrDefault(root);
47             var leftSize = GetSizeOrZero(left);
48             if (LessThan(index, leftSize))
49             {
50                 root = left;
51                 continue;
52             }
53             if (IsEquals(index, leftSize))
54             {
55                 return root;
56             }
57             root = GetRightOrDefault(root);
58             index = Subtract(index, Increment(leftSize));
59         }
60         return GetZero(); // TODO: Impossible situation exception (only if tree
        ↳ structure broken)
61     }
62 }
63
64 // TODO: Return indices range instead of references count
65 public TLink CountUsages(TLink link)
66 {
67     var root = GetTreeRoot();
68     var total = GetSize(root);
69     var totalRightIgnore = GetZero();
70     while (!EqualToZero(root))
71     {
72         var @base = GetBasePartValue(root);
73         if (LessOrEqualThan(@base, link))
74         {
75             root = GetRightOrDefault(root);
76         }
77         else
78         {
79             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
80             root = GetLeftOrDefault(root);
81         }
82     }
83     root = GetTreeRoot();
84     var totalLeftIgnore = GetZero();
85     while (!EqualToZero(root))
86     {
87         var @base = GetBasePartValue(root);
88         if (GreaterOrEqualThan(@base, link))
89         {
90             root = GetLeftOrDefault(root);
91         }
92         else
93         {
94             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
95             root = GetRightOrDefault(root);
96         }
97     }
98     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
99 }
100
101 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
102 {
103     var root = GetTreeRoot();
104     if (EqualToZero(root))
105     {
106         return _constants.Continue;
107     }
108 }

```

```

109     TLink first = GetZero(), current = root;
110     while (!EqualToZero(current))
111     {
112         var @base = GetBasePartValue(current);
113         if (GreaterOrEqualThan(@base, link))
114         {
115             if (IsEquals(@base, link))
116             {
117                 first = current;
118             }
119             current = GetLeftOrDefault(current);
120         }
121         else
122         {
123             current = GetRightOrDefault(current);
124         }
125     }
126     if (!EqualToZero(first))
127     {
128         current = first;
129         while (true)
130         {
131             if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
132             {
133                 return _constants.Break;
134             }
135             current = GetNext(current);
136             if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
137             {
138                 break;
139             }
140         }
141     }
142     return _constants.Continue;
143 }
144
145 protected override void PrintNodeValue(TLink node, StringBuilder sb)
146 {
147     sb.Append(' ');
148     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
149 ↪ Link.SourceOffset).GetValue<TLink>());
150     sb.Append('-');
151     sb.Append('>');
152     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
153 ↪ Link.TargetOffset).GetValue<TLink>());
154 }
155
156 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
157 {
158     public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
159         : base(memory)
160     {
161     }
162
163     protected override IntPtr GetLeftPointer(TLink node) =>
164         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset;
165
166     protected override IntPtr GetRightPointer(TLink node) =>
167         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset;
168
169     protected override TLink GetLeftValue(TLink node) =>
170         ↪ (Links.GetElement(LinkSizeInBytes, node) +
171 ↪ Link.LeftAsSourceOffset).GetValue<TLink>();
172
173     protected override TLink GetRightValue(TLink node) =>
174         ↪ (Links.GetElement(LinkSizeInBytes, node) +
175 ↪ Link.RightAsSourceOffset).GetValue<TLink>();
176
177     protected override TLink GetSize(TLink node)
178     {
179         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
180 ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
181         return Bit.PartialRead(previousValue, 5, -5);
182     }
183 }

```

```

176     protected override void SetLeft(TLink node, TLink left) =>
177         ↪ (Links.GetElement(LinkSizeInBytes, node) +
178         ↪ Link.LeftAsSourceOffset).SetValue(left);

179
180     protected override void SetRight(TLink node, TLink right) =>
181         ↪ (Links.GetElement(LinkSizeInBytes, node) +
182         ↪ Link.RightAsSourceOffset).SetValue(right);

183
184     protected override void SetSize(TLink node, TLink size)
185     {
186         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
187         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
188         (Links.GetElement(LinkSizeInBytes, node) +
189         ↪ Link.SizeAsSourceOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
190         ↪ -5));
191     }

192
193     protected override bool GetLeftIsChild(TLink node)
194     {
195         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
196         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
197         return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
198     }

199
200     protected override void SetLeftIsChild(TLink node, bool value)
201     {
202         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
203         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
204         var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
205         ↪ 1);
206         (Links.GetElement(LinkSizeInBytes, node) +
207         ↪ Link.SizeAsSourceOffset).SetValue(modified);
208     }

209
210     protected override bool GetRightIsChild(TLink node)
211     {
212         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
213         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
214         return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
215     }

216
217     protected override void SetRightIsChild(TLink node, bool value)
218     {
219         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
220         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
221         var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
222         ↪ 1);
223         (Links.GetElement(LinkSizeInBytes, node) +
224         ↪ Link.SizeAsSourceOffset).SetValue(modified);
225     }

226
227     protected override sbyte GetBalance(TLink node)
228     {
229         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
230         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
231         var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
232         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
233         ↪ 124 : value & 3);
234         return unpackedValue;
235     }

236
237     protected override void SetBalance(TLink node, sbyte value)
238     {
239         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
240         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
241         var packagedValue = (TLink)(Integer<TLink>)(((byte)value >> 5) & 4) | value &
242         ↪ 3);
243         var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
244         (Links.GetElement(LinkSizeInBytes, node) +
245         ↪ Link.SizeAsSourceOffset).SetValue(modified);
246     }

247
248     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
249     {
250         var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
251         ↪ Link.SourceOffset).GetValue<TLink>();

```

```

231     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
232         ↪ Link.SourceOffset).GetValue<TLink>();
233     return LessThan(firstSource, secondSource) ||
234         (IsEquals(firstSource, secondSource) &&
235             ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
236                 ↪ Link.TargetOffset).GetValue<TLink>(),
237                 ↪ (Links.GetElement(LinkSizeInBytes, second) +
238                 ↪ Link.TargetOffset).GetValue<TLink>()));
239 }
240
241 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
242 {
243     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
244         ↪ Link.SourceOffset).GetValue<TLink>();
245     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
246         ↪ Link.SourceOffset).GetValue<TLink>();
247     return GreaterThan(firstSource, secondSource) ||
248         (IsEquals(firstSource, secondSource) &&
249             ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
250                 ↪ Link.TargetOffset).GetValue<TLink>(),
251                 ↪ (Links.GetElement(LinkSizeInBytes, second) +
252                 ↪ Link.TargetOffset).GetValue<TLink>()));
253 }
254
255 protected override TLink GetTreeRoot() => (Header +
256     ↪ LinksHeader.FirstAsSourceOffset).GetValue<TLink>();
257
258 protected override TLink GetBasePartValue(TLink link) =>
259     ↪ (Links.GetElement(LinkSizeInBytes, link) + Link.SourceOffset).GetValue<TLink>();
260
261 /// <summary>
262 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
263     ↪ (концом)
264 /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
265 /// </summary>
266 /// <param name="source">Индекс связи, которая является началом на искомой
267     ↪ связи.</param>
268 /// <param name="target">Индекс связи, которая является концом на искомой
269     ↪ связи.</param>
270 /// <returns>Индекс искомой связи.</returns>
271 public TLink Search(TLink source, TLink target)
272 {
273     var root = GetTreeRoot();
274     while (!EqualToZero(root))
275     {
276         var rootSource = (Links.GetElement(LinkSizeInBytes, root) +
277             ↪ Link.SourceOffset).GetValue<TLink>();
278         var rootTarget = (Links.GetElement(LinkSizeInBytes, root) +
279             ↪ Link.TargetOffset).GetValue<TLink>();
280         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
281             ↪ node.Key < root.Key
282         {
283             root = GetLeftOrDefault(root);
284         }
285         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
286             ↪ // node.Key > root.Key
287         {
288             root = GetRightOrDefault(root);
289         }
290         else // node.Key == root.Key
291         {
292             return root;
293         }
294     }
295     return GetZero();
296 }
297
298 [MethodImpl(MethodImplOptions.AggressiveInlining)]
299 private bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget, TLink
300     ↪ secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
301     ↪ (IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
302
303 [MethodImpl(MethodImplOptions.AggressiveInlining)]
304 private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget, TLink
305     ↪ secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
306     ↪ (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));

```

```

283 }
284
285 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
286 {
287     public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
288         : base(memory)
289     {
290     }
291
292     protected override IntPtr GetLeftPointer(TLink node) =>
293         ↳ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset;
294
295     protected override IntPtr GetRightPointer(TLink node) =>
296         ↳ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset;
297
298     protected override TLink GetLeftValue(TLink node) =>
299         ↳ (Links.GetElement(LinkSizeInBytes, node) +
300         ↳ Link.LeftAsTargetOffset).GetValue<TLink>();
301
302     protected override TLink GetRightValue(TLink node) =>
303         ↳ (Links.GetElement(LinkSizeInBytes, node) +
304         ↳ Link.RightAsTargetOffset).GetValue<TLink>();
305
306     protected override TLink GetSize(TLink node)
307     {
308         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
309         ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
310         return Bit.PartialRead(previousValue, 5, -5);
311     }
312
313     protected override void SetLeft(TLink node, TLink left) =>
314         ↳ (Links.GetElement(LinkSizeInBytes, node) +
315         ↳ Link.LeftAsTargetOffset).SetValue(left);
316
317     protected override void SetRight(TLink node, TLink right) =>
318         ↳ (Links.GetElement(LinkSizeInBytes, node) +
319         ↳ Link.RightAsTargetOffset).SetValue(right);
320
321     protected override void SetSize(TLink node, TLink size)
322     {
323         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
324         ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
325         (Links.GetElement(LinkSizeInBytes, node) +
326         ↳ Link.SizeAsTargetOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
327         ↳ -5));
328     }
329
330     protected override bool GetLeftIsChild(TLink node)
331     {
332         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
333         ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
334         return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
335     }
336
337     protected override void SetLeftIsChild(TLink node, bool value)
338     {
339         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
340         ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
341         var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
342         ↳ 1);
343         (Links.GetElement(LinkSizeInBytes, node) +
344         ↳ Link.SizeAsTargetOffset).SetValue(modified);
345     }
346
347     protected override bool GetRightIsChild(TLink node)
348     {
349         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
350         ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
351         return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
352     }
353
354     protected override void SetRightIsChild(TLink node, bool value)
355     {
356         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
357         ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
358         var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
359         ↳ 1);

```

```

339         (Links.GetElement(LinkSizeInBytes, node) +
        ↪ Link.SizeAsTargetOffset).SetValue(modified);
340     }
341
342     protected override sbyte GetBalance(TLink node)
343     {
344         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
345         var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
346         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
        ↪ 124 : value & 3);
347         return unpackedValue;
348     }
349
350     protected override void SetBalance(TLink node, sbyte value)
351     {
352         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
353         var packagedValue = (TLink)(Integer<TLink>)(((byte)value >> 5) & 4) | value &
        ↪ 3);
354         var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
355         (Links.GetElement(LinkSizeInBytes, node) +
        ↪ Link.SizeAsTargetOffset).SetValue(modified);
356     }
357
358     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
359     {
360         var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
        ↪ Link.TargetOffset).GetValue<TLink>();
361         var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
        ↪ Link.TargetOffset).GetValue<TLink>();
362         return LessThan(firstTarget, secondTarget) ||
363             (IsEquals(firstTarget, secondTarget) &&
        ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
        ↪ Link.SourceOffset).GetValue<TLink>(),
        ↪ (Links.GetElement(LinkSizeInBytes, second) +
        ↪ Link.SourceOffset).GetValue<TLink>()));
364     }
365
366     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
367     {
368         var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
        ↪ Link.TargetOffset).GetValue<TLink>();
369         var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
        ↪ Link.TargetOffset).GetValue<TLink>();
370         return GreaterThan(firstTarget, secondTarget) ||
371             (IsEquals(firstTarget, secondTarget) &&
        ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
        ↪ Link.SourceOffset).GetValue<TLink>(),
        ↪ (Links.GetElement(LinkSizeInBytes, second) +
        ↪ Link.SourceOffset).GetValue<TLink>()));
372     }
373
374     protected override TLink GetTreeRoot() => (Header +
        ↪ LinksHeader.FirstAsTargetOffset).GetValue<TLink>();
375
376     protected override TLink GetBasePartValue(TLink link) =>
        ↪ (Links.GetElement(LinkSizeInBytes, link) + Link.TargetOffset).GetValue<TLink>();
377 }
378 }
379 }

```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs

```

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

```

```

16 // ReSharper disable BuiltInTypeReferenceStyle
17
18 namespace Platform.Data.Doublets.ResizableDirectMemory
19 {
20     using id = UInt64;
21
22     public unsafe partial class UInt64ResizableDirectMemoryLinks : DisposableBase, ILinks<id>
23     {
24         /// <summary>Возвращает размер одной связи в байтах.</summary>
25         /// <remarks>
26         /// Используется только во вне класса, не рекомендуется использовать внутри.
27         /// Так как во вне не обязательно будет доступен unsafe C#.
28         /// </remarks>
29         public static readonly int LinkSizeInBytes = sizeof(Link);
30
31         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
32
33         private struct Link
34         {
35             public id Source;
36             public id Target;
37             public id LeftAsSource;
38             public id RightAsSource;
39             public id SizeAsSource;
40             public id LeftAsTarget;
41             public id RightAsTarget;
42             public id SizeAsTarget;
43         }
44
45         private struct LinksHeader
46         {
47             public id AllocatedLinks;
48             public id ReservedLinks;
49             public id FreeLinks;
50             public id FirstFreeLink;
51             public id FirstAsSource;
52             public id FirstAsTarget;
53             public id LastFreeLink;
54             public id Reserved8;
55         }
56
57         private readonly long _memoryReservationStep;
58
59         private readonly IResizableDirectMemory _memory;
60         private LinksHeader* _header;
61         private Link* _links;
62
63         private LinksTargetsTreeMethods _targetsTreeMethods;
64         private LinksSourcesTreeMethods _sourcesTreeMethods;
65
66         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
67         // → нужно использовать не список а дерево, так как так можно быстрее проверить на
68         // → наличие связи внутри
69         private UnusedLinksListMethods _unusedLinksListMethods;
70
71         /// <summary>
72         /// Возвращает общее число связей находящихся в хранилище.
73         /// </summary>
74         private id Total => _header->AllocatedLinks - _header->FreeLinks;
75
76         // TODO: Дать возможность переопределять в конструкторе
77         public LinksCombinedConstants<id, id, int> Constants { get; }
78
79         public UInt64ResizableDirectMemoryLinks(string address) : this(address,
80             → DefaultLinksSizeStep) { }
81
82         /// <summary>
83         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
84         /// → минимальным шагом расширения базы данных.
85         /// </summary>
86         /// <param name="address">Полный путь к файлу базы данных.</param>
87         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
88         /// → байтах.</param>
89         public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
90             → this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
91             → memoryReservationStep) { }
92
93         public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
94             → DefaultLinksSizeStep) { }
95
96     }
97 }

```



```

88 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep)
89 {
90     Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
91     _memory = memory;
92     _memoryReservationStep = memoryReservationStep;
93     if (memory.ReservedCapacity < memoryReservationStep)
94     {
95         memory.ReservedCapacity = memoryReservationStep;
96     }
97     SetPointers(_memory);
98     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
99     _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) +
    ↳ sizeof(LinksHeader);
100    // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
101    _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(LinksHeader)) /
    ↳ sizeof(Link));
102 }
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 public id Count(IList<id> restrictions)
106 {
107     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
108     if (restrictions.Count == 0)
109     {
110         return Total;
111     }
112     if (restrictions.Count == 1)
113     {
114         var index = restrictions[Constants.IndexPart];
115         if (index == Constants.Any)
116         {
117             return Total;
118         }
119         return Exists(index) ? 1UL : 0UL;
120     }
121     if (restrictions.Count == 2)
122     {
123         var index = restrictions[Constants.IndexPart];
124         var value = restrictions[1];
125         if (index == Constants.Any)
126         {
127             if (value == Constants.Any)
128             {
129                 return Total; // Any - как отсутствие ограничения
130             }
131             return _sourcesTreeMethods.CountUsages(value)
132                 + _targetsTreeMethods.CountUsages(value);
133         }
134         else
135         {
136             if (!Exists(index))
137             {
138                 return 0;
139             }
140             if (value == Constants.Any)
141             {
142                 return 1;
143             }
144             var storedLinkValue = GetLinkUnsafe(index);
145             if (storedLinkValue->Source == value ||
146                 storedLinkValue->Target == value)
147             {
148                 return 1;
149             }
150             return 0;
151         }
152     }
153     if (restrictions.Count == 3)
154     {
155         var index = restrictions[Constants.IndexPart];
156         var source = restrictions[Constants.SourcePart];
157         var target = restrictions[Constants.TargetPart];
158         if (index == Constants.Any)
159         {
160             if (source == Constants.Any && target == Constants.Any)
161             {
162                 return Total;

```

```

163     }
164     else if (source == Constants.Any)
165     {
166         return _targetsTreeMethods.CountUsages(target);
167     }
168     else if (target == Constants.Any)
169     {
170         return _sourcesTreeMethods.CountUsages(source);
171     }
172     else //if(source != Any && target != Any)
173     {
174         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
175         var link = _sourcesTreeMethods.Search(source, target);
176         return link == Constants.Null ? OUL : 1UL;
177     }
178 }
179 else
180 {
181     if (!Exists(index))
182     {
183         return 0;
184     }
185     if (source == Constants.Any && target == Constants.Any)
186     {
187         return 1;
188     }
189     var storedLinkValue = GetLinkUnsafe(index);
190     if (source != Constants.Any && target != Constants.Any)
191     {
192         if (storedLinkValue->Source == source &&
193             storedLinkValue->Target == target)
194         {
195             return 1;
196         }
197         return 0;
198     }
199     var value = default(id);
200     if (source == Constants.Any)
201     {
202         value = target;
203     }
204     if (target == Constants.Any)
205     {
206         value = source;
207     }
208     if (storedLinkValue->Source == value ||
209         storedLinkValue->Target == value)
210     {
211         return 1;
212     }
213     return 0;
214 }
215 }
216 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
217 }
218
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
221 {
222     if (restrictions.Count == 0)
223     {
224         for (id link = 1; link <= _header->AllocatedLinks; link++)
225         {
226             if (Exists(link))
227             {
228                 if (handler(GetLinkStruct(link)) == Constants.Break)
229                 {
230                     return Constants.Break;
231                 }
232             }
233         }
234         return Constants.Continue;
235     }
236     if (restrictions.Count == 1)
237     {
238         var index = restrictions[Constants.IndexPart];
239         if (index == Constants.Any)
240         {

```

```

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

```

```

318     }
319     var storedLinkValue = GetLinkUnsafe(index);
320     if (source != Constants.Any && target != Constants.Any)
321     {
322         if (storedLinkValue->Source == source &&
323             storedLinkValue->Target == target)
324         {
325             return handler(GetLinkStruct(index));
326         }
327         return Constants.Continue;
328     }
329     var value = default(id);
330     if (source == Constants.Any)
331     {
332         value = target;
333     }
334     if (target == Constants.Any)
335     {
336         value = source;
337     }
338     if (storedLinkValue->Source == value ||
339         storedLinkValue->Target == value)
340     {
341         return handler(GetLinkStruct(index));
342     }
343     return Constants.Continue;
344 }
345 }
346 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
347 }
348
349 /// <remarks>
350 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
351 /// ↳ в другом месте (но не в менеджере памяти, а в логике Links)
352 /// </remarks>
353 [MethodImpl(MethodImplOptions.AggressiveInlining)]
354 public id Update(IList<id> values)
355 {
356     var linkIndex = values[Constants.IndexPart];
357     var link = GetLinkUnsafe(linkIndex);
358     // Будет корректно работать только в том случае, если пространство выделенной связи
359     ↳ предварительно заполнено нулями
360     if (link->Source != Constants.Null)
361     {
362         _sourcesTreeMethods.Detach(new IntPtr(&_header->FirstAsSource), linkIndex);
363     }
364     if (link->Target != Constants.Null)
365     {
366         _targetsTreeMethods.Detach(new IntPtr(&_header->FirstAsTarget), linkIndex);
367     }
368 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
369     var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
370     var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
371     if (leftTreeSize != rightTreeSize)
372     {
373         throw new Exception("One of the trees is broken.");
374     }
375 #endif
376     link->Source = values[Constants.SourcePart];
377     link->Target = values[Constants.TargetPart];
378     if (link->Source != Constants.Null)
379     {
380         _sourcesTreeMethods.Attach(new IntPtr(&_header->FirstAsSource), linkIndex);
381     }
382     if (link->Target != Constants.Null)
383     {
384         _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
385     }
386 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
387     leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
388     rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
389     if (leftTreeSize != rightTreeSize)
390     {
391         throw new Exception("One of the trees is broken.");
392     }
393 #endif
394     return linkIndex;

```

```

393 }
394
395 [MethodImpl(MethodImplOptions.AggressiveInlining)]
396 private IList<id> GetLinkStruct(id linkIndex)
397 {
398     var link = GetLinkUnsafe(linkIndex);
399     return new UInt64Link(linkIndex, link->Source, link->Target);
400 }
401
402 [MethodImpl(MethodImplOptions.AggressiveInlining)]
403 private Link* GetLinkUnsafe(id linkIndex) => &_amp;_links[linkIndex];
404
405 /// <remarks>
406 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
407   ↳ пространство
408 /// </remarks>
409 public id Create()
410 {
411     var freeLink = _header->FirstFreeLink;
412     if (freeLink != Constants.Null)
413     {
414         _unusedLinksListMethods.Detach(freeLink);
415     }
416     else
417     {
418         if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
419         {
420             throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
421         }
422         if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
423         {
424             _memory.ReservedCapacity += _memory.ReservationStep;
425             SetPointers(_memory);
426             _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
427         }
428         _header->AllocatedLinks++;
429         _memory.UsedCapacity += sizeof(Link);
430         freeLink = _header->AllocatedLinks;
431     }
432     return freeLink;
433 }
434
435 public void Delete(id link)
436 {
437     if (link < _header->AllocatedLinks)
438     {
439         _unusedLinksListMethods.AttachAsFirst(link);
440     }
441     else if (link == _header->AllocatedLinks)
442     {
443         _header->AllocatedLinks--;
444         _memory.UsedCapacity -= sizeof(Link);
445         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
446         // пока не дойдём до первой существующей связи
447         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
448         while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
449         {
450             _unusedLinksListMethods.Detach(_header->AllocatedLinks);
451             _header->AllocatedLinks--;
452             _memory.UsedCapacity -= sizeof(Link);
453         }
454     }
455 }
456
457 /// <remarks>
458 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
459   ↳ адрес реально поменялся
460 ///
461 /// Указатель this.links может быть в том же месте,
462 /// так как 0-я связь не используется и имеет такой же размер как Header,
463 /// поэтому header размещается в том же месте, что и 0-я связь
464 /// </remarks>
465 private void SetPointers(IResizableDirectMemory memory)
466 {
467     if (memory == null)
468     {
469         _header = null;
470         _links = null;
471         _unusedLinksListMethods = null;
472     }
473 }

```

```

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

```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs

```

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

```

```
39 }
```

```
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs
```

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

```

76         {
77             totalRightIgnore += GetRightSize(root) + 1;
78             root = GetLeftOrDefault(root);
79         }
80     }
81     root = GetTreeRoot();
82     var totalLeftIgnore = 0UL;
83     while (root != 0)
84     {
85         var @base = GetBasePartValue(root);
86         if (@base >= link)
87         {
88             root = GetLeftOrDefault(root);
89         }
90         else
91         {
92             totalLeftIgnore += GetLeftSize(root) + 1;
93             root = GetRightOrDefault(root);
94         }
95     }
96     return total - totalRightIgnore - totalLeftIgnore;
97 }
98
99 public ulong EachReference(ulong link, Func<IList<ulong>, ulong> handler)
100 {
101     var root = GetTreeRoot();
102     if (root == 0)
103     {
104         return _constants.Continue;
105     }
106     ulong first = 0, current = root;
107     while (current != 0)
108     {
109         var @base = GetBasePartValue(current);
110         if (@base >= link)
111         {
112             if (@base == link)
113             {
114                 first = current;
115             }
116             current = GetLeftOrDefault(current);
117         }
118         else
119         {
120             current = GetRightOrDefault(current);
121         }
122     }
123     if (first != 0)
124     {
125         current = first;
126         while (true)
127         {
128             if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
129             {
130                 return _constants.Break;
131             }
132             current = GetNext(current);
133             if (current == 0 || GetBasePartValue(current) != link)
134             {
135                 break;
136             }
137         }
138     }
139     return _constants.Continue;
140 }
141
142 protected override void PrintNodeValue(ulong node, StringBuilder sb)
143 {
144     sb.Append(' ');
145     sb.Append(Links[node].Source);
146     sb.Append('-');
147     sb.Append('>');
148     sb.Append(Links[node].Target);
149 }
150
151 }
152
153 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
154 {
155     public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)

```



```

155         : base(memory)
156     {
157     }
158
159     protected override IntPtr GetLeftPointer(ulong node) => new
160     ↪ IntPtr(&Links[node].LeftAsSource);
161
162     protected override IntPtr GetRightPointer(ulong node) => new
163     ↪ IntPtr(&Links[node].RightAsSource);
164
165     protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsSource;
166
167     protected override ulong GetRightValue(ulong node) => Links[node].RightAsSource;
168
169     protected override ulong GetSize(ulong node)
170     {
171         var previousValue = Links[node].SizeAsSource;
172         //return Math.PartialRead(previousValue, 5, -5);
173         return (previousValue & 4294967264) >> 5;
174     }
175
176     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource
177     ↪ = left;
178
179     protected override void SetRight(ulong node, ulong right) =>
180     ↪ Links[node].RightAsSource = right;
181
182     protected override void SetSize(ulong node, ulong size)
183     {
184         var previousValue = Links[node].SizeAsSource;
185         //var modified = Math.PartialWrite(previousValue, size, 5, -5);
186         var modified = (previousValue & 31) | ((size & 134217727) << 5);
187         Links[node].SizeAsSource = modified;
188     }
189
190     protected override bool GetLeftIsChild(ulong node)
191     {
192         var previousValue = Links[node].SizeAsSource;
193         //return (Integer)Math.PartialRead(previousValue, 4, 1);
194         return (previousValue & 16) >> 4 == 1UL;
195     }
196
197     protected override void SetLeftIsChild(ulong node, bool value)
198     {
199         var previousValue = Links[node].SizeAsSource;
200         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
201         var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
202         Links[node].SizeAsSource = modified;
203     }
204
205     protected override bool GetRightIsChild(ulong node)
206     {
207         var previousValue = Links[node].SizeAsSource;
208         //return (Integer)Math.PartialRead(previousValue, 3, 1);
209         return (previousValue & 8) >> 3 == 1UL;
210     }
211
212     protected override void SetRightIsChild(ulong node, bool value)
213     {
214         var previousValue = Links[node].SizeAsSource;
215         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
216         var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
217         Links[node].SizeAsSource = modified;
218     }
219
220     protected override sbyte GetBalance(ulong node)
221     {
222         var previousValue = Links[node].SizeAsSource;
223         //var value = Math.PartialRead(previousValue, 0, 3);
224         var value = previousValue & 7;
225         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
226         ↪ 124 : value & 3);
227         return unpackedValue;
228     }
229
230     protected override void SetBalance(ulong node, sbyte value)
231     {
232         var previousValue = Links[node].SizeAsSource;
233         var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);

```

```

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

```

```

296     protected override ulong GetOne() => 1UL;
297
298     [MethodImpl(MethodImplOptions.AggressiveInlining)]
299     protected override ulong GetTwo() => 2UL;
300
301     [MethodImpl(MethodImplOptions.AggressiveInlining)]
302     protected override bool ValueEqualToZero(IntPtr pointer) =>
303         ↪ *(ulong*)pointer.ToPointer() == 0UL;
304
305     [MethodImpl(MethodImplOptions.AggressiveInlining)]
306     protected override bool EqualToZero(ulong value) => value == 0UL;
307
308     [MethodImpl(MethodImplOptions.AggressiveInlining)]
309     protected override bool IsEquals(ulong first, ulong second) => first == second;
310
311     [MethodImpl(MethodImplOptions.AggressiveInlining)]
312     protected override bool GreaterThanZero(ulong value) => value > 0UL;
313
314     [MethodImpl(MethodImplOptions.AggressiveInlining)]
315     protected override bool GreaterThan(ulong first, ulong second) => first > second;
316
317     [MethodImpl(MethodImplOptions.AggressiveInlining)]
318     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >=
319         ↪ second;
320
321     [MethodImpl(MethodImplOptions.AggressiveInlining)]
322     protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0
323         ↪ is always true for ulong
324
325     [MethodImpl(MethodImplOptions.AggressiveInlining)]
326     protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is
327         ↪ always >= 0 for ulong
328
329     [MethodImpl(MethodImplOptions.AggressiveInlining)]
330     protected override bool LessOrEqualThan(ulong first, ulong second) => first <=
331         ↪ second;
332
333     [MethodImpl(MethodImplOptions.AggressiveInlining)]
334     protected override bool LessThanZero(ulong value) => false; // value < 0 is always
335         ↪ false for ulong
336
337     [MethodImpl(MethodImplOptions.AggressiveInlining)]
338     protected override bool LessThan(ulong first, ulong second) => first < second;
339
340     [MethodImpl(MethodImplOptions.AggressiveInlining)]
341     protected override ulong Increment(ulong value) => ++value;
342
343     [MethodImpl(MethodImplOptions.AggressiveInlining)]
344     protected override ulong Decrement(ulong value) => --value;
345
346     [MethodImpl(MethodImplOptions.AggressiveInlining)]
347     protected override ulong Add(ulong first, ulong second) => first + second;
348
349     [MethodImpl(MethodImplOptions.AggressiveInlining)]
350     protected override ulong Subtract(ulong first, ulong second) => first - second;
351 }
352
353 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
354 {
355     public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
356         : base(memory)
357     {
358     }
359
360     //protected override IntPtr GetLeft(ulong node) => new
361     ↪ IntPtr(&Links[node].LeftAsTarget);
362
363     //protected override IntPtr GetRight(ulong node) => new
364     ↪ IntPtr(&Links[node].RightAsTarget);
365
366     //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
367
368     //protected override void SetLeft(ulong node, ulong left) =>
369     ↪ Links[node].LeftAsTarget = left;
370
371     //protected override void SetRight(ulong node, ulong right) =>
372     ↪ Links[node].RightAsTarget = right;
373

```

```

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

```

```

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

```

./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs

```

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

```

```

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

```

# ./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs

```

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

```

```

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

```

```

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

```



```

203 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
204 {
205     var frequency = data.Frequency;
206     var maxFrequency = _maxDoubletData.Frequency;
207     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
    ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
    ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
    ↪ _maxDoublet.Target)))
208 if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
209     (_comparer.Compare(maxFrequency, frequency) < 0 ||
    ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
    ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
    ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
    ↪ better stability and better compression on sequent data and even on random
    ↪ numbers data (but gives collisions anyway) */
210 {
211     _maxDoublet = doublet;
212     _maxDoubletData = data;
213 }
214 }
215 }
216 }

```

./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs

```

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

```

./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs

```

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

```

```

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

```

./Platform.Data.Doublets/Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs

```

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

```

```

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

```

./Platform.Data.Doublets/Sequences/CreteriaMatchers/DefaultSequenceElementCriterionMatcher.cs

```

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

```

./Platform.Data.Doublets/Sequences/CreteriaMatchers/MarkedSequenceCriterionMatcher.cs

```

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

```

./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs

```

1 using System.Collections.Generic;
2 using Platform.Collections.Stacks;
3 using Platform.Data.Doublets.Sequences.HeightProviders;
4 using Platform.Data.Sequences;
5
6 namespace Platform.Data.Doublets.Sequences
7 {
8     public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
    ↪ ISequenceAppender<TLink>
9     {

```

```

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

```

#### ./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs

```

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

```

#### ./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs

```

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

```

```

16 {
17     private readonly ILinks<TLink> _links;
18     private readonly ISequences<TLink> _sequences;
19     private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
20     private BitString _visited;
21
22     private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
23     ↪ IList<TLink>>>
24     {
25         private readonly IListEqualityComparer<TLink> _listComparer;
26         public ItemEquilityComparer() => _listComparer =
27             ↪ Default<IListEqualityComparer<TLink>>.Instance;
28         public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
29             ↪ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
30             ↪ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
31             ↪ right.Value);
32         public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
33             ↪ (_listComparer.GetHashCode(pair.Key),
34             ↪ _listComparer.GetHashCode(pair.Value)).GetHashCode();
35     }
36
37     private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
38     {
39         private readonly IListComparer<TLink> _listComparer;
40
41         public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
42
43         public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
44             ↪ KeyValuePair<IList<TLink>, IList<TLink>> right)
45         {
46             var intermediateResult = _listComparer.Compare(left.Key, right.Key);
47             if (intermediateResult == 0)
48             {
49                 intermediateResult = _listComparer.Compare(left.Value, right.Value);
50             }
51             return intermediateResult;
52         }
53     }
54
55     public DuplicateSegmentsProvider(ILinks<TLink> links, ISequences<TLink> sequences)
56     : base(minimumStringSegmentLength: 2)
57     {
58         _links = links;
59         _sequences = sequences;
60     }
61
62     public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
63     {
64         _groups = new HashSet<KeyValuePair<IList<TLink>,
65             ↪ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
66         var count = _links.Count();
67         _visited = new BitString((long)(Integer<TLink>)count + 1);
68         _links.Each(link =>
69         {
70             var linkIndex = _links.GetIndex(link);
71             var linkBitIndex = (long)(Integer<TLink>)linkIndex;
72             if (!_visited.Get(linkBitIndex))
73             {
74                 var sequenceElements = new List<TLink>();
75                 _sequences.EachPart(sequenceElements.AddAndReturnTrue, linkIndex);
76                 if (sequenceElements.Count > 2)
77                 {
78                     WalkAll(sequenceElements);
79                 }
80             }
81             return _links.Constants.Continue;
82         });
83         var resultList = _groups.ToList();
84         var comparer = Default<ItemComparer>.Instance;
85         resultList.Sort(comparer);
86
87         #if DEBUG
88         foreach (var item in resultList)
89         {
90             PrintDuplicates(item);
91         }
92         #endif
93         return resultList;
94     }
95 }
96

```

```

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

```

./Platform.Data.Doublets/Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs

```

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

```

```

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

```

./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs

```

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

```

```

57 public void PrintFrequencies(ICollection<TLink> sequence)
58 {
59     for (var i = 1; i < sequence.Count; i++)
60     {
61         PrintFrequency(sequence[i - 1], sequence[i]);
62     }
63 }
64
65 public void PrintFrequency(TLink source, TLink target)
66 {
67     var number = GetFrequency(source, target).Frequency;
68     Console.WriteLine("{0},{1} - {2}", source, target, number);
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
73 {
74     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
75     {
76         data.IncrementFrequency();
77     }
78     else
79     {
80         var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
81         data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
82         if (!_equalityComparer.Equals(link, default))
83         {
84             data.Frequency = Arithmetic.Add(data.Frequency,
85                 ↪ _frequencyCounter.Count(link));
86         }
87         _doubletsCache.Add(doublet, data);
88     }
89     return data;
90 }
91
92 public void ValidateFrequencies()
93 {
94     foreach (var entry in _doubletsCache)
95     {
96         var value = entry.Value;
97         var linkIndex = value.Link;
98         if (!_equalityComparer.Equals(linkIndex, default))
99         {
100             var frequency = value.Frequency;
101             var count = _frequencyCounter.Count(linkIndex);
102             // TODO: Why `frequency` always greater than `count` by 1?
103             if (((_comparer.Compare(frequency, count) > 0) &&
104                 ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count),
105                 ↪ Integer<TLink>.One) > 0))
106                 || ((_comparer.Compare(count, frequency) > 0) &&
107                 ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency),
108                 ↪ Integer<TLink>.One) > 0)))
109             {
110                 throw new InvalidOperationException("Frequencies validation failed.");
111             }
112             //else
113             //{
114             //    if (value.Frequency > 0)
115             //    {
116             //        var frequency = value.Frequency;
117             //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
118             //        var count = _countLinkFrequency(linkIndex);
119             //        if ((frequency > count && frequency - count > 1) || (count > frequency
120             //            ↪ && count - frequency > 1))
121             //            throw new Exception("Frequencies validation failed.");
122             //    }
123             //}
124     }
125 }

```

./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Numbers;
3

```



```

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

```

./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.cs

```

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

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs

```

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

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3 using Platform.Numbers;
4 using Platform.Data.Sequences;
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7 {
8     public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11         ⇨ EqualityComparer<TLink>.Default;
12     }
13 }

```

```

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

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs

```

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

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter

```

1 using Platform.Interfaces;
2 using Platform.Numbers;
3
4 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
5 {
6     public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
7         ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
8     {
9         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
10
11        public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
12            ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
13            : base(links, symbol)
14            => _markedSequenceMatcher = markedSequenceMatcher;
15    }
16 }

```

```

13
14     protected override void CountSequenceSymbolFrequency(TLink link)
15     {
16         var symbolFrequencyCounter = new
            ↳ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
            ↳ _markedSequenceMatcher, link, _symbol);
17         _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
18     }
19 }
20 }

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs

```

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

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3 using Platform.Numbers;
4
5 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6 {
7     public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
8     {
9         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
10        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11
12        protected readonly ILinks<TLink> _links;
13        protected readonly TLink _symbol;
14        protected readonly HashSet<TLink> _visits;
15        protected TLink _total;
16
17        public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
18        {
19            _links = links;
20            _symbol = symbol;
21            _visits = new HashSet<TLink>();
22            _total = default;
23        }
24
25        public TLink Count()
26        {
27            if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
28            {
29                return _total;
30            }
31            CountCore(_symbol);
32            return _total;
33        }
34
35        private void CountCore(TLink link)
36        {
37            var any = _links.Constants.Any;
38            if (_equalityComparer.Equals(_links.Count(any, link), default))
39            {
40                CountSequenceSymbolFrequency(link);
41            }
42            else
43            {
44                _links.Each(EachElementHandler, any, link);
45            }
46        }
47
48        protected virtual void CountSequenceSymbolFrequency(TLink link)
49        {
50            var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
            ↳ link, _symbol);
51            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
52        }

```

```

53
54     private TLink EachElementHandler(IList<TLink> doublet)
55     {
56         var constants = _links.Constants;
57         var doubletIndex = doublet[constants.IndexPart];
58         if (_visits.Add(doubletIndex))
59         {
60             CountCore(doubletIndex);
61         }
62         return constants.Continue;
63     }
64 }
65 }

```

./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Sequences.HeightProviders
5  {
6      public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
7          ↳ ISequenceHeightProvider<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↳ EqualityComparer<TLink>.Default;
11
12         private readonly TLink _heightPropertyMarker;
13         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
14         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
15         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
16         private readonly IPropertiesOperator<TLink, TLink, TLink> _propertyOperator;
17
18         public CachedSequenceHeightProvider(
19             ILinks<TLink> links,
20             ISequenceHeightProvider<TLink> baseHeightProvider,
21             IConverter<TLink> addressToUnaryNumberConverter,
22             IConverter<TLink> unaryNumberToAddressConverter,
23             TLink heightPropertyMarker,
24             IPropertiesOperator<TLink, TLink, TLink> propertyOperator)
25             : base(links)
26         {
27             _heightPropertyMarker = heightPropertyMarker;
28             _baseHeightProvider = baseHeightProvider;
29             _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
30             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
31             _propertyOperator = propertyOperator;
32         }
33
34         public TLink Get(TLink sequence)
35         {
36             TLink height;
37             var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
38             if (_equalityComparer.Equals(heightValue, default))
39             {
40                 height = _baseHeightProvider.Get(sequence);
41                 heightValue = _addressToUnaryNumberConverter.Convert(height);
42                 _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
43             }
44             else
45             {
46                 height = _unaryNumberToAddressConverter.Convert(heightValue);
47             }
48             return height;
49         }
50     }
51 }

```

./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```

1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  namespace Platform.Data.Doublets.Sequences.HeightProviders
5  {
6      public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
7          ↳ ISequenceHeightProvider<TLink>
8      {
9          private readonly ICriterionMatcher<TLink> _elementMatcher;
10
11         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
12             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
13     }
14 }

```

```

11
12     public TLink Get(TLink sequence)
13     {
14         var height = default(TLink);
15         var pairOrElement = sequence;
16         while (!_elementMatcher.IsMatched(pairOrElement))
17         {
18             pairOrElement = Links.GetTarget(pairOrElement);
19             height = Arithmetic.Increment(height);
20         }
21         return height;
22     }
23 }
24 }

```

./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs

```

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

```

./Platform.Data.Doublets/Sequences/Sequences.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Runtime.CompilerServices;
5 using Platform.Collections;
6 using Platform.Collections.Lists;
7 using Platform.Threading.Synchronization;
8 using Platform.Singletons;
9 LinkIndex = System.UInt64;
10 using Platform.Data.Constants;
11 using Platform.Data.Sequences;
12 using Platform.Data.Doublets.Sequences.Walkers;
13 using Platform.Collections.Stacks;
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 (подпоследовательностей),
26     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
27     /// вместе, все числа вместе и т.п.
28     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
29     /// графа)
30     ///
31     /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
32     /// ограничитель на то, что является последовательностью, а что нет,
33     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
34     /// порядке.
35     ///
36     /// Рост последовательности слева и справа.
37     /// Поиск со звёздочкой.
38     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
39     /// так же проблема может быть решена при реализации дистанционных триггеров.
40     /// Нужны ли уникальные указатели вообще?
41     /// Что если обращение к информации будет происходить через содержимое всегда?
42     ///
43     /// Писать тесты.
44     ///
45     /// Можно убрать зависимость от конкретной реализации Links,
46     /// на зависимость от абстрактного элемента, который может быть представлен несколькими
47     /// способами.
48     ///
49     /// Можно ли как-то сделать один общий интерфейс
50     ///
51     /// Блокчейн и/или гит для распределённой записи транзакций.

```

```

49  ///
50  /// </remarks>
51  public partial class Sequences : ISequences<ulong> // IList<string>, IList<ulong[]> (после
    ↳ завершения реализации Sequences)
52  {
53      private static readonly LinksCombinedConstants<bool, ulong, long> _constants =
        ↳ Default<LinksCombinedConstants<bool, ulong, long>>.Instance;
54
55      /// <summary>Возвращает значение ulong, обозначающее любое количество связей.</summary>
56      public const ulong ZeroOrMany = ulong.MaxValue;
57
58      public SequencesOptions<ulong> Options;
59      public readonly SynchronizedLinks<ulong> Links;
60      public readonly ISynchronization Sync;
61
62      public Sequences(SynchronizedLinks<ulong> links)
63          : this(links, new SequencesOptions<ulong>())
64      {
65      }
66
67      public Sequences(SynchronizedLinks<ulong> links, SequencesOptions<ulong> options)
68      {
69          Links = links;
70          Sync = links.SyncRoot;
71          Options = options;
72
73          Options.ValidateOptions();
74          Options.InitOptions(Links);
75      }
76
77      public bool IsSequence(ulong sequence)
78      {
79          return Sync.ExecuteReadOperation(() =>
80          {
81              if (Options.UseSequenceMarker)
82              {
83                  return Options.MarkedSequenceMatcher.IsMatched(sequence);
84              }
85              return !Links.Unsync.IsPartialPoint(sequence);
86          });
87      }
88
89      [MethodImpl(MethodImplOptions.AggressiveInlining)]
90      private ulong GetSequenceByElements(ulong sequence)
91      {
92          if (Options.UseSequenceMarker)
93          {
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, _constants.Any);
123         }
124         if (sequence.Length == 1) // Первая связь это адрес
125         {

```

```

126         if (sequence[0] == _constants.Null)
127         {
128             return 0;
129         }
130         if (sequence[0] == _constants.Any)
131         {
132             return Count();
133         }
134         if (Options.UseSequenceMarker)
135         {
136             return Links.Count(_constants.Any, Options.SequenceMarkerLink, sequence[0]);
137         }
138         return Links.Exists(sequence[0]) ? 1UL : 0;
139     }
140     throw new NotImplementedException();
141 }
142
143 private ulong CountUsages(params ulong[] restrictions)
144 {
145     if (restrictions.Length == 0)
146     {
147         return 0;
148     }
149     if (restrictions.Length == 1) // Первая связь это адрес
150     {
151         if (restrictions[0] == _constants.Null)
152         {
153             return 0;
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
170 #endregion
171
172 #region Create
173
174 public ulong Create(params ulong[] sequence)
175 {
176     return Sync.ExecuteWriteOperation(() =>
177     {
178         if (sequence.IsNullOrEmpty())
179         {
180             return _constants.Null;
181         }
182         Links.EnsureEachLinkExists(sequence);
183         return CreateCore(sequence);
184     });
185 }
186
187 private ulong CreateCore(params ulong[] sequence)
188 {
189     if (Options.UseIndex)
190     {
191         Options.Indexer.Index(sequence);
192     }
193     var sequenceRoot = default(ulong);
194     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
195     {
196         var matches = Each(sequence);
197         if (matches.Count > 0)
198         {
199             sequenceRoot = matches[0];
200         }
201     }
202     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
203     {
204         return CompactCore(sequence);
205     }
206 }

```

```

205     }
206     if (sequenceRoot == default)
207     {
208         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
209     }
210     if (Options.UseSequenceMarker)
211     {
212         Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
213     }
214     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
215 }
216
217 #endregion
218
219 #region Each
220
221 public List<ulong> Each(params ulong[] sequence)
222 {
223     var results = new List<ulong>();
224     Each(results.AddAndReturnTrue, sequence);
225     return results;
226 }
227
228 public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
229 {
230     return Sync.ExecuteReadOperation(() =>
231     {
232         if (sequence.IsNullOrEmpty())
233         {
234             return true;
235         }
236         Links.EnsureEachLinkIsAnyOrExists(sequence);
237         if (sequence.Count == 1)
238         {
239             var link = sequence[0];
240             if (link == _constants.Any)
241             {
242                 return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
243             }
244             return handler(link);
245         }
246         if (sequence.Count == 2)
247         {
248             return Links.Unsync.Each(sequence[0], sequence[1], handler);
249         }
250         if (Options.UseIndex && !Options.Indexer.CheckIndex(sequence))
251         {
252             return false;
253         }
254         return EachCore(handler, sequence);
255     });
256 }
257
258 private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
259 {
260     var matcher = new Matcher(this, sequence, new HashSet<LinkIndex>(), handler);
261     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
262     ↪ Id.
263     Func<ulong, bool> innerHandler = Options.UseSequenceMarker ? (Func<ulong,
264     ↪ bool>)matcher.HandleFullMatchedSequence : matcher.HandleFullMatched;
265     //if (sequence.Length >= 2)
266     if (!StepRight(innerHandler, sequence[0], sequence[1]))
267     {
268         return false;
269     }
270     var last = sequence.Count - 2;
271     for (var i = 1; i < last; i++)
272     {
273         if (!PartialStepRight(innerHandler, sequence[i], sequence[i + 1]))
274         {
275             return false;
276         }
277     }
278     if (sequence.Count >= 3)
279     {
280         if (!StepLeft(innerHandler, sequence[sequence.Count - 2],
281         ↪ sequence[sequence.Count - 1]))
282         {
283             return false;
284         }
285     }
286 }

```



```

281     }
282 }
283 return true;
284 }
285
286 private bool PartialStepRight(Func<ulong, bool> handler, ulong left, ulong right)
287 {
288     return Links.Unsync.Each(_constants.Any, left, doublet =>
289     {
290         if (!StepRight(handler, doublet, right))
291         {
292             return false;
293         }
294         if (left != doublet)
295         {
296             return PartialStepRight(handler, doublet, right);
297         }
298         return true;
299     });
300 }
301
302 private bool StepRight(Func<ulong, bool> handler, ulong left, ulong right) =>
303     ↳ Links.Unsync.Each(left, _constants.Any, rightStep => TryStepRightUp(handler, right,
304     ↳ rightStep));
305
306 private bool TryStepRightUp(Func<ulong, bool> handler, ulong right, ulong stepFrom)
307 {
308     var upStep = stepFrom;
309     var firstSource = Links.Unsync.GetTarget(upStep);
310     while (firstSource != right && firstSource != upStep)
311     {
312         upStep = firstSource;
313         firstSource = Links.Unsync.GetSource(upStep);
314     }
315     if (firstSource == right)
316     {
317         return handler(stepFrom);
318     }
319     return true;
320 }
321
322 private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong right) =>
323     ↳ Links.Unsync.Each(_constants.Any, right, leftStep => TryStepLeftUp(handler, left,
324     ↳ leftStep));
325
326 private bool TryStepLeftUp(Func<ulong, bool> handler, ulong left, ulong stepFrom)
327 {
328     var upStep = stepFrom;
329     var firstTarget = Links.Unsync.GetSource(upStep);
330     while (firstTarget != left && firstTarget != upStep)
331     {
332         upStep = firstTarget;
333         firstTarget = Links.Unsync.GetTarget(upStep);
334     }
335     if (firstTarget == left)
336     {
337         return handler(stepFrom);
338     }
339     return true;
340 }
341
342 #endregion
343
344 #region Update
345
346 public ulong Update(ulong[] sequence, ulong[] newSequence)
347 {
348     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
349     {
350         return _constants.Null;
351     }
352     if (sequence.IsNullOrEmpty())
353     {
354         return Create(newSequence);
355     }
356     if (newSequence.IsNullOrEmpty())
357     {
358         Delete(sequence);
359         return _constants.Null;
360     }
361 }

```

```

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 последовательности и с
379     ↪ маркером,
380     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
381     ↪ 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 {
394     if (Options.UseGarbageCollection)
395     {
396         var sequenceElements = GetSequenceElements(sequence);
397         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
398         var newSequenceLink = GetSequenceByElements(sequenceElements);
399         var newSequenceElements = GetSequenceElements(newSequence);
400         var newSequenceLink = GetSequenceByElements(newSequenceElements);
401         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
402         {
403             if (sequenceLink != _constants.Null)
404             {
405                 Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
406             }
407             Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
408         }
409         ClearGarbage(sequenceElementsContents.Source);
410         ClearGarbage(sequenceElementsContents.Target);
411     }
412     else
413     {
414         if (Options.UseSequenceMarker)
415         {
416             var sequenceElements = GetSequenceElements(sequence);
417             var sequenceLink = GetSequenceByElements(sequenceElements);
418             var newSequenceElements = GetSequenceElements(newSequence);
419             var newSequenceLink = GetSequenceByElements(newSequenceElements);
420             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
421             {
422                 if (sequenceLink != _constants.Null)
423                 {
424                     Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
425                 }
426                 Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
427             }
428         }
429         else
430         {
431             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
432             {
433                 Links.Unsync.MergeUsages(sequence, newSequence);
434             }
435         }
436     }
437 }

```

```

431     }
432 }
433 }
434 }
435
436 #endregion
437
438 #region Delete
439
440 public void Delete(params ulong[] sequence)
441 {
442     Sync.ExecuteWriteOperation(() =>
443     {
444         // TODO: Check all options only ones before loop execution
445         foreach (var linkToDelete in Each(sequence))
446         {
447             DeleteOneCore(linkToDelete);
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 UInt64Link(Links.GetLink(sequenceElements));
458         var sequenceLink = GetSequenceByElements(sequenceElements);
459         if (Options.UseCascadeDelete || CountUsages(link) == 0)
460         {
461             if (sequenceLink != _constants.Null)
462             {
463                 Links.Unsync.Delete(sequenceLink);
464             }
465             Links.Unsync.Delete(link);
466         }
467         ClearGarbage(sequenceElementsContents.Source);
468         ClearGarbage(sequenceElementsContents.Target);
469     }
470     else
471     {
472         if (Options.UseSequenceMarker)
473         {
474             var sequenceElements = GetSequenceElements(link);
475             var sequenceLink = GetSequenceByElements(sequenceElements);
476             if (Options.UseCascadeDelete || CountUsages(link) == 0)
477             {
478                 if (sequenceLink != _constants.Null)
479                 {
480                     Links.Unsync.Delete(sequenceLink);
481                 }
482                 Links.Unsync.Delete(link);
483             }
484         }
485         else
486         {
487             if (Options.UseCascadeDelete || CountUsages(link) == 0)
488             {
489                 Links.Unsync.Delete(link);
490             }
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 возможность,
502 /// гарантировать его использование в других местах).
503 ///
504 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
505 /// </remarks>
506 public ulong Compact(params ulong[] sequence)
507 {
508     return Sync.ExecuteWriteOperation(() =>
509     {

```

```

510         if (sequence.IsNullOrEmpty())
511         {
512             return _constants.Null;
513         }
514         Links.EnsureEachLinkExists(sequence);
515         return CompactCore(sequence);
516     });
517 }
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
528 ///         ↳ определить извне или в унаследованном классе
529 /// </remarks>
530 [MethodImpl(MethodImplOptions.AggressiveInlining)]
531 private bool IsGarbage(ulong link) => link != Options.SequenceMarkerLink &&
532     ↳ !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;
533
534 private void ClearGarbage(ulong link)
535 {
536     if (IsGarbage(link))
537     {
538         var contents = new UInt64Link(Links.GetLink(link));
539         Links.Unsync.Delete(link);
540         ClearGarbage(contents.Source);
541         ClearGarbage(contents.Target);
542     }
543 }
544
545 #endregion
546
547 #region Walkers
548
549 public bool EachPart(Func<ulong, bool> handler, ulong sequence)
550 {
551     return Sync.ExecuteReadOperation(() =>
552     {
553         var links = Links.Unsync;
554         var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>());
555         foreach (var part in walker.Walk(sequence))
556         {
557             if (!handler(links.GetIndex(part)))
558             {
559                 return false;
560             }
561         }
562         return true;
563     });
564 }
565
566 public class Matcher : RightSequenceWalker<ulong>
567 {
568     private readonly Sequences _sequences;
569     private readonly IList<LinkIndex> _patternSequence;
570     private readonly HashSet<LinkIndex> _linksInSequence;
571     private readonly HashSet<LinkIndex> _results;
572     private readonly Func<ulong, bool> _stopableHandler;
573     private readonly HashSet<ulong> _readAsElements;
574     private int _filterPosition;
575
576     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
577         ↳ HashSet<LinkIndex> results, Func<LinkIndex, bool> stopableHandler,
578         ↳ HashSet<LinkIndex> readAsElements = null)
579         : base(sequences.Links.Unsync, new DefaultStack<ulong>())
580     {
581         _sequences = sequences;
582         _patternSequence = patternSequence;
583         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
584             ↳ _constants.Any && x != ZeroOrMany));
585         _results = results;
586         _stopableHandler = stopableHandler;
587         _readAsElements = readAsElements;
588     }

```

```

585     protected override bool IsElement(IList<ulong> link) => base.IsElement(link) ||
    ↪     (_readAsElements != null && _readAsElements.Contains(Links.GetIndex(link))) ||
    ↪     _linksInSequence.Contains(Links.GetIndex(link));

586
587     public bool FullMatch(LinkIndex sequenceToMatch)
588     {
589         _filterPosition = 0;
590         foreach (var part in Walk(sequenceToMatch))
591         {
592             if (!FullMatchCore(Links.GetIndex(part)))
593             {
594                 break;
595             }
596         }
597         return _filterPosition == _patternSequence.Count;
598     }
599
600     private bool FullMatchCore(LinkIndex element)
601     {
602         if (_filterPosition == _patternSequence.Count)
603         {
604             _filterPosition = -2; // Длиннее чем нужно
605             return false;
606         }
607         if (_patternSequence[_filterPosition] != _constants.Any
608             && element != _patternSequence[_filterPosition])
609         {
610             _filterPosition = -1;
611             return false; // Начинается/Продолжается иначе
612         }
613         _filterPosition++;
614         return true;
615     }
616
617     public void AddFullMatchedToResults(ulong sequenceToMatch)
618     {
619         if (FullMatch(sequenceToMatch))
620         {
621             _results.Add(sequenceToMatch);
622         }
623     }
624
625     public bool HandleFullMatched(ulong sequenceToMatch)
626     {
627         if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
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     /// <remarks>
645     /// TODO: Add support for LinksConstants.Any
646     /// </remarks>
647     public bool PartialMatch(LinkIndex sequenceToMatch)
648     {
649         _filterPosition = -1;
650         foreach (var part in Walk(sequenceToMatch))
651         {
652             if (!PartialMatchCore(Links.GetIndex(part)))
653             {
654                 break;
655             }
656         }
657         return _filterPosition == _patternSequence.Count - 1;
658     }
659
660     private bool PartialMatchCore(LinkIndex element)

```

```

661     {
662         if (_filterPosition == (_patternSequence.Count - 1))
663         {
664             return false; // Нашлось
665         }
666         if (_filterPosition >= 0)
667         {
668             if (element == _patternSequence[_filterPosition + 1])
669             {
670                 _filterPosition++;
671             }
672             else
673             {
674                 _filterPosition = -1;
675             }
676         }
677         if (_filterPosition < 0)
678         {
679             if (element == _patternSequence[0])
680             {
681                 _filterPosition = 0;
682             }
683         }
684         return true; // Ищем дальше
685     }
686
687     public void AddPartialMatchedToResults(ulong sequenceToMatch)
688     {
689         if (PartialMatch(sequenceToMatch))
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 }

```

./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs

```

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

```

```

7 using Platform.Collections;
8 using Platform.Data.Exceptions;
9 using Platform.Data.Sequences;
10 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
11 using Platform.Data.Doublets.Sequences.Walkers;
12 using Platform.Collections.Stacks;
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     partial class Sequences
17     {
18         #region Create All Variants (Not Practical)
19
20         /// <remarks>
21         /// Number of links that is needed to generate all variants for
22         /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
23         /// </remarks>
24         public ulong[] CreateAllVariants2(ulong[] sequence)
25         {
26             return Sync.ExecuteWriteOperation(() =>
27             {
28                 if (sequence.IsNullOrEmpty())
29                 {
30                     return new ulong[0];
31                 }
32                 Links.EnsureEachLinkExists(sequence);
33                 if (sequence.Length == 1)
34                 {
35                     return sequence;
36                 }
37                 return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
38             });
39         }
40
41         private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
42         {
43             #if DEBUG
44                 if ((stopAt - startAt) < 0)
45                 {
46                     throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
47                         ↪ меньше или равен stopAt");
48                 }
49             #endif
50
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], sequence[stopAt])
58                     ↪ };
59             }
60
61             var variants = new ulong[(ulong)Numbers.Math.Catalan(stopAt - startAt)];
62             var last = 0;
63             for (var splitter = startAt; splitter < stopAt; splitter++)
64             {
65                 var left = CreateAllVariants2Core(sequence, startAt, splitter);
66                 var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
67                 for (var i = 0; i < left.Length; i++)
68                 {
69                     for (var j = 0; j < right.Length; j++)
70                     {
71                         var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
72                         if (variant == _constants.Null)
73                         {
74                             throw new NotImplementedException("Creation cancellation is not
75                                 ↪ implemented.");
76                         }
77                         variants[last++] = variant;
78                     }
79                 }
80             }
81             return variants;
82         }
83
84         public List<ulong> CreateAllVariants1(params ulong[] sequence)
85         {
86             return Sync.ExecuteWriteOperation(() =>
87             {
88

```

```

83         if (sequence.IsNullOrEmpty())
84         {
85             return new List<ulong>();
86         }
87         Links.Unsync.EnsureEachLinkExists(sequence);
88         if (sequence.Length == 1)
89         {
90             return new List<ulong> { sequence[0] };
91         }
92         var results = new List<ulong>((int)Numbers.Math.Catalan(sequence.Length));
93         return CreateAllVariants1Core(sequence, results);
94     });
95 }
96
97 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
98 {
99     if (sequence.Length == 2)
100     {
101         var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
102         if (link == _constants.Null)
103         {
104             throw new NotImplementedException("Creation cancellation is not
105                 ↳ implemented.");
106         }
107         results.Add(link);
108         return results;
109     }
110     var innerSequenceLength = sequence.Length - 1;
111     var innerSequence = new ulong[innerSequenceLength];
112     for (var li = 0; li < innerSequenceLength; li++)
113     {
114         var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
115         if (link == _constants.Null)
116         {
117             throw new NotImplementedException("Creation cancellation is not
118                 ↳ implemented.");
119         }
120         for (var isi = 0; isi < li; isi++)
121         {
122             innerSequence[isi] = sequence[isi];
123         }
124         innerSequence[li] = link;
125         for (var isi = li + 1; isi < innerSequenceLength; isi++)
126         {
127             innerSequence[isi] = sequence[isi + 1];
128         }
129         CreateAllVariants1Core(innerSequence, results);
130     }
131     return results;
132 }
133
134 #endregion
135
136 public HashSet<ulong> Each1(params ulong[] sequence)
137 {
138     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
139     Each1(link =>
140     {
141         if (!visitedLinks.Contains(link))
142         {
143             visitedLinks.Add(link); // изучить почему случаются повторы
144         }
145         return true;
146     }, sequence);
147     return visitedLinks;
148 }
149
150 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
151 {
152     if (sequence.Length == 2)
153     {
154         Links.Unsync.Each(sequence[0], sequence[1], handler);
155     }
156     else
157     {
158         var innerSequenceLength = sequence.Length - 1;
159         for (var li = 0; li < innerSequenceLength; li++)
160         {

```



```

159     var left = sequence[li];
160     var right = sequence[li + 1];
161     if (left == 0 && right == 0)
162     {
163         continue;
164     }
165     var linkIndex = li;
166     ulong[] innerSequence = null;
167     Links.Unsync.Each(left, right, doublet =>
168     {
169         if (innerSequence == null)
170         {
171             innerSequence = new ulong[innerSequenceLength];
172             for (var isi = 0; isi < linkIndex; isi++)
173             {
174                 innerSequence[isi] = sequence[isi];
175             }
176             for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
177             {
178                 innerSequence[isi] = sequence[isi + 1];
179             }
180         }
181         innerSequence[linkIndex] = doublet;
182         Each1(handler, innerSequence);
183         return _constants.Continue;
184     });
185 }
186 }
187 }
188
189 public HashSet<ulong> EachPart(params ulong[] sequence)
190 {
191     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
192     EachPartCore(link =>
193     {
194         if (!visitedLinks.Contains(link))
195         {
196             visitedLinks.Add(link); // изучить почему случаются повторы
197         }
198         return true;
199     }, sequence);
200     return visitedLinks;
201 }
202
203 public void EachPart(Func<ulong, bool> handler, params ulong[] sequence)
204 {
205     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
206     EachPartCore(link =>
207     {
208         if (!visitedLinks.Contains(link))
209         {
210             visitedLinks.Add(link); // изучить почему случаются повторы
211             return handler(link);
212         }
213         return true;
214     }, sequence);
215 }
216
217 private void EachPartCore(Func<ulong, bool> handler, params ulong[] sequence)
218 {
219     if (sequence.IsNullOrEmpty())
220     {
221         return;
222     }
223     Links.EnsureEachLinkIsAnyOrExists(sequence);
224     if (sequence.Length == 1)
225     {
226         var link = sequence[0];
227         if (link > 0)
228         {
229             handler(link);
230         }
231         else
232         {
233             Links.Each(_constants.Any, _constants.Any, handler);
234         }
235     }
236     else if (sequence.Length == 2)
237     {

```

```

238 // _links.Each(sequence[0], sequence[1], handler);
239 //   o_|      x_o ...
240 // x_|      |___|
241 Links.Each(sequence[1], _constants.Any, doublet =>
242 {
243     var match = Links.SearchOrDefault(sequence[0], doublet);
244     if (match != _constants.Null)
245     {
246         handler(match);
247     }
248     return true;
249 });
250 // |_x      ... x_o
251 // |_o      |___|
252 Links.Each(_constants.Any, sequence[0], doublet =>
253 {
254     var match = Links.SearchOrDefault(doublet, sequence[1]);
255     if (match != 0)
256     {
257         handler(match);
258     }
259     return true;
260 });
261 //      .x o_.
262 //      |___|
263 PartialStepRight(x => handler(x), sequence[0], sequence[1]);
264 }
265 else
266 {
267     // TODO: Implement other variants
268     return;
269 }
270 }
271
272 private void PartialStepRight(Action<ulong> handler, ulong left, ulong right)
273 {
274     Links.Unsync.Each(_constants.Any, left, doublet =>
275     {
276         StepRight(handler, doublet, right);
277         if (left != doublet)
278         {
279             PartialStepRight(handler, doublet, right);
280         }
281         return true;
282     });
283 }
284
285 private void StepRight(Action<ulong> handler, ulong left, ulong right)
286 {
287     Links.Unsync.Each(left, _constants.Any, rightStep =>
288     {
289         TryStepRightUp(handler, right, rightStep);
290         return true;
291     });
292 }
293
294 private void TryStepRightUp(Action<ulong> handler, ulong right, ulong stepFrom)
295 {
296     var upStep = stepFrom;
297     var firstSource = Links.Unsync.GetTarget(upStep);
298     while (firstSource != right && firstSource != upStep)
299     {
300         upStep = firstSource;
301         firstSource = Links.Unsync.GetSource(upStep);
302     }
303     if (firstSource == right)
304     {
305         handler(stepFrom);
306     }
307 }
308
309 // TODO: Test
310 private void PartialStepLeft(Action<ulong> handler, ulong left, ulong right)
311 {
312     Links.Unsync.Each(right, _constants.Any, doublet =>
313     {
314         StepLeft(handler, left, doublet);
315         if (right != doublet)
316         {

```

```

317         PartialStepLeft(handler, left, doublet);
318     }
319     return true;
320 });
321 }
322
323 private void StepLeft(Action<ulong> handler, ulong left, ulong right)
324 {
325     Links.Unsync.Each(_constants.Any, right, leftStep =>
326     {
327         TryStepLeftUp(handler, left, leftStep);
328         return true;
329     });
330 }
331
332 private void TryStepLeftUp(Action<ulong> handler, ulong left, ulong stepFrom)
333 {
334     var upStep = stepFrom;
335     var firstTarget = Links.Unsync.GetSource(upStep);
336     while (firstTarget != left && firstTarget != upStep)
337     {
338         upStep = firstTarget;
339         firstTarget = Links.Unsync.GetTarget(upStep);
340     }
341     if (firstTarget == left)
342     {
343         handler(stepFrom);
344     }
345 }
346
347 private bool StartsWith(ulong sequence, ulong link)
348 {
349     var upStep = sequence;
350     var firstSource = Links.Unsync.GetSource(upStep);
351     while (firstSource != link && firstSource != upStep)
352     {
353         upStep = firstSource;
354         firstSource = Links.Unsync.GetSource(upStep);
355     }
356     return firstSource == link;
357 }
358
359 private bool EndsWith(ulong sequence, ulong link)
360 {
361     var upStep = sequence;
362     var lastTarget = Links.Unsync.GetTarget(upStep);
363     while (lastTarget != link && lastTarget != upStep)
364     {
365         upStep = lastTarget;
366         lastTarget = Links.Unsync.GetTarget(upStep);
367     }
368     return lastTarget == link;
369 }
370
371 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
372 {
373     return Sync.ExecuteReadOperation(() =>
374     {
375         var results = new List<ulong>();
376         if (sequence.Length > 0)
377         {
378             Links.EnsureEachLinkExists(sequence);
379             var firstElement = sequence[0];
380             if (sequence.Length == 1)
381             {
382                 results.Add(firstElement);
383                 return results;
384             }
385             if (sequence.Length == 2)
386             {
387                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
388                 if (doublet != _constants.Null)
389                 {
390                     results.Add(doublet);
391                 }
392                 return results;
393             }
394             var linksInSequence = new HashSet<ulong>(sequence);
395             void handler(ulong result)

```

```

396     {
397         var filterPosition = 0;
398         StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
399             ↪ Links.Unsync.GetTarget,
400             x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
401             ↪ x =>
402             {
403                 if (filterPosition == sequence.Length)
404                 {
405                     filterPosition = -2; // Длиннее чем нужно
406                     return false;
407                 }
408                 if (x != sequence[filterPosition])
409                 {
410                     filterPosition = -1;
411                     return false; // Начинается иначе
412                 }
413                 filterPosition++;
414                 return true;
415             });
416         if (filterPosition == sequence.Length)
417         {
418             results.Add(result);
419         }
420     }
421     if (sequence.Length >= 2)
422     {
423         StepRight(handler, sequence[0], sequence[1]);
424     }
425     var last = sequence.Length - 2;
426     for (var i = 1; i < last; i++)
427     {
428         PartialStepRight(handler, sequence[i], sequence[i + 1]);
429     }
430     if (sequence.Length >= 3)
431     {
432         StepLeft(handler, sequence[sequence.Length - 2],
433             ↪ sequence[sequence.Length - 1]);
434     }
435 }
436 return results;
437 });
438 }
439
440 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
441 {
442     return Sync.ExecuteReadOperation(() =>
443     {
444         var results = new HashSet<ulong>();
445         if (sequence.Length > 0)
446         {
447             Links.EnsureEachLinkExists(sequence);
448             var firstElement = sequence[0];
449             if (sequence.Length == 1)
450             {
451                 results.Add(firstElement);
452                 return results;
453             }
454             if (sequence.Length == 2)
455             {
456                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
457                 if (doublet != _constants.Null)
458                 {
459                     results.Add(doublet);
460                 }
461                 return results;
462             }
463             var matcher = new Matcher(this, sequence, results, null);
464             if (sequence.Length >= 2)
465             {
466                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
467             }
468             var last = sequence.Length - 2;
469             for (var i = 1; i < last; i++)
470             {
471                 PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
472                     ↪ sequence[i + 1]);

```

```

470     }
471     if (sequence.Length >= 3)
472     {
473         StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
474             ↪ sequence[sequence.Length - 1]);
475     }
476     return results;
477 });
478 }
479
480 public const int MaxSequenceFormatSize = 200;
481
482 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
483     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
484
485 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
486     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
487     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
488     ↪ elementToString, insertComma, knownElements));
489
490 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
491     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
492     ↪ LinkIndex[] knownElements)
493 {
494     var linksInSequence = new HashSet<ulong>(knownElements);
495     //var entered = new HashSet<ulong>();
496     var sb = new StringBuilder();
497     sb.Append('{');
498     if (links.Exists(sequenceLink))
499     {
500         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
501             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
502             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
503         {
504             if (insertComma && sb.Length > 1)
505             {
506                 sb.Append(',');
507             }
508             //if (entered.Contains(element))
509             //{
510             //    sb.Append('{');
511             //    elementToString(sb, element);
512             //    sb.Append('}');
513             //}
514             //else
515             elementToString(sb, element);
516             if (sb.Length < MaxSequenceFormatSize)
517             {
518                 return true;
519             }
520             sb.Append(insertComma ? ", ..." : "...");
521             return false;
522         });
523     }
524     sb.Append('}');
525     return sb.ToString();
526 }
527
528 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
529     ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
530     ↪ knownElements);
531
532 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
533     ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
534     ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
535     ↪ sequenceLink, elementToString, insertComma, knownElements));
536
537 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
538     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
539     ↪ LinkIndex[] knownElements)
540 {
541     var linksInSequence = new HashSet<ulong>(knownElements);
542     var entered = new HashSet<ulong>();
543     var sb = new StringBuilder();
544     sb.Append('{');
545     if (links.Exists(sequenceLink))
546     {

```

```

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

```

```

608         filteredResults.Add(result);
609     }
610 }
611     return filteredResults;
612 }
613     return new List<ulong>();
614 });
615 }
616
617 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
618 {
619     return Sync.ExecuteReadOperation(() =>
620     {
621         if (sequence.Length > 0)
622         {
623             Links.EnsureEachLinkExists(sequence);
624             var results = new HashSet<ulong>();
625             for (var i = 0; i < sequence.Length; i++)
626             {
627                 AllUsagesCore(sequence[i], results);
628             }
629             var filteredResults = new HashSet<ulong>();
630             var matcher = new Matcher(this, sequence, filteredResults, null);
631             matcher.AddAllPartialMatchedToResults(results);
632             return filteredResults;
633         }
634         return new HashSet<ulong>();
635     });
636 }
637
638 public bool GetAllPartiallyMatchingSequences2(Func<ulong, bool> handler, params ulong[]
639 → sequence)
640 {
641     return Sync.ExecuteReadOperation(() =>
642     {
643         if (sequence.Length > 0)
644         {
645             Links.EnsureEachLinkExists(sequence);
646
647             var results = new HashSet<ulong>();
648             var filteredResults = new HashSet<ulong>();
649             var matcher = new Matcher(this, sequence, filteredResults, handler);
650             for (var i = 0; i < sequence.Length; i++)
651             {
652                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
653                 {
654                     return false;
655                 }
656             }
657             return true;
658         }
659         return true;
660     });
661 }
662
663 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
664 //{
665 //    return Sync.ExecuteReadOperation(() =>
666 //    {
667 //        if (sequence.Length > 0)
668 //        {
669 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
670
671 //            var firstResults = new HashSet<ulong>();
672 //            var lastResults = new HashSet<ulong>();
673
674 //            var first = sequence.First(x => x != LinksConstants.Any);
675 //            var last = sequence.Last(x => x != LinksConstants.Any);
676
677 //            AllUsagesCore(first, firstResults);
678 //            AllUsagesCore(last, lastResults);
679
680 //            firstResults.IntersectWith(lastResults);
681
682 //            //for (var i = 0; i < sequence.Length; i++)
683 //            //    AllUsagesCore(sequence[i], results);
684
685 //            var filteredResults = new HashSet<ulong>();
686 //            var matcher = new Matcher(this, sequence, filteredResults, null);

```

```

686 //         matcher.AddAllPartialMatchedToResults(firstResults);
687 //         return filteredResults;
688 //     }
689
690 //     return new HashSet<ulong>();
691 // });
692 //}
693
694 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
695 {
696     return Sync.ExecuteReadOperation(() =>
697     {
698         if (sequence.Length > 0)
699         {
700             Links.EnsureEachLinkIsAnyOrExists(sequence);
701             var firstResults = new HashSet<ulong>();
702             var lastResults = new HashSet<ulong>();
703             var first = sequence.First(x => x != _constants.Any);
704             var last = sequence.Last(x => x != _constants.Any);
705             AllUsagesCore(first, firstResults);
706             AllUsagesCore(last, lastResults);
707             firstResults.IntersectWith(lastResults);
708             //for (var i = 0; i < sequence.Length; i++)
709             //    AllUsagesCore(sequence[i], results);
710             var filteredResults = new HashSet<ulong>();
711             var matcher = new Matcher(this, sequence, filteredResults, null);
712             matcher.AddAllPartialMatchedToResults(firstResults);
713             return filteredResults;
714         }
715         return new HashSet<ulong>();
716     });
717 }
718
719 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
720 ↪ IList<ulong> sequence)
721 {
722     return Sync.ExecuteReadOperation(() =>
723     {
724         if (sequence.Count > 0)
725         {
726             Links.EnsureEachLinkExists(sequence);
727             var results = new HashSet<LinkIndex>();
728             //var nextResults = new HashSet<ulong>();
729             //for (var i = 0; i < sequence.Length; i++)
730             //{
731             //    AllUsagesCore(sequence[i], nextResults);
732             //    if (results.IsNullOrEmpty())
733             //    {
734             //        results = nextResults;
735             //        nextResults = new HashSet<ulong>();
736             //    }
737             //    else
738             //    {
739             //        results.IntersectWith(nextResults);
740             //        nextResults.Clear();
741             //    }
742             //}
743             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
744             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
745             var next = new HashSet<ulong>();
746             for (var i = 1; i < sequence.Count; i++)
747             {
748                 var collector = new AllUsagesCollector1(Links.Unsync, next);
749                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
750
751                 results.IntersectWith(next);
752                 next.Clear();
753             }
754             var filteredResults = new HashSet<ulong>();
755             var matcher = new Matcher(this, sequence, filteredResults, null,
756 ↪ readAsElements);
757             matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
758 ↪ x)); // OrderBy is a Hack
759             return filteredResults;
760         }
761         return new HashSet<ulong>();
762     });
763 }

```



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

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

```

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

```

```

905         AllBottomUsagesCore(link, visits, usages);
906         return usages;
907     });
908 }
909
910 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
↪ usages)
911 {
912     bool handler(ulong doublet)
913     {
914         if (visits.Add(doublet))
915         {
916             AllBottomUsagesCore(doublet, visits, usages);
917         }
918         return true;
919     }
920     if (Links.Unsync.Count(_constants.Any, link) == 0)
921     {
922         usages.Add(link);
923     }
924     else
925     {
926         Links.Unsync.Each(link, _constants.Any, handler);
927         Links.Unsync.Each(_constants.Any, link, handler);
928     }
929 }
930
931 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
932 {
933     if (Options.UseSequenceMarker)
934     {
935         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
↪ Options.MarkedSequenceMatcher, symbol);
936         return counter.Count();
937     }
938     else
939     {
940         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
↪ symbol);
941         return counter.Count();
942     }
943 }
944
945 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool>
↪ outerHandler)
946 {
947     bool handler(ulong doublet)
948     {
949         if (usages.Add(doublet))
950         {
951             if (!outerHandler(doublet))
952             {
953                 return false;
954             }
955             if (!AllUsagesCore1(doublet, usages, outerHandler))
956             {
957                 return false;
958             }
959         }
960         return true;
961     }
962     return Links.Unsync.Each(link, _constants.Any, handler)
963         && Links.Unsync.Each(_constants.Any, link, handler);
964 }
965
966 public void CalculateAllUsages(ulong[] totals)
967 {
968     var calculator = new AllUsagesCalculator(Links, totals);
969     calculator.Calculate();
970 }
971
972 public void CalculateAllUsages2(ulong[] totals)
973 {
974     var calculator = new AllUsagesCalculator2(Links, totals);
975     calculator.Calculate();
976 }
977
978 private class AllUsagesCalculator

```

```

979 {
980     private readonly SynchronizedLinks<ulong> _links;
981     private readonly ulong[] _totals;
982
983     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
984     {
985         _links = links;
986         _totals = totals;
987     }
988
989     public void Calculate() => _links.Each(_constants.Any, _constants.Any,
990         ↪ CalculateCore);
991
992     private bool CalculateCore(ulong link)
993     {
994         if (_totals[link] == 0)
995         {
996             var total = 1UL;
997             _totals[link] = total;
998             var visitedChildren = new HashSet<ulong>();
999             bool linkCalculator(ulong child)
1000             {
1001                 if (link != child && visitedChildren.Add(child))
1002                 {
1003                     total += _totals[child] == 0 ? 1 : _totals[child];
1004                 }
1005                 return true;
1006             }
1007             _links.Unsync.Each(link, _constants.Any, linkCalculator);
1008             _links.Unsync.Each(_constants.Any, link, linkCalculator);
1009             _totals[link] = total;
1010         }
1011         return true;
1012     }
1013 }
1014
1015 private class AllUsagesCalculator2
1016 {
1017     private readonly SynchronizedLinks<ulong> _links;
1018     private readonly ulong[] _totals;
1019
1020     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1021     {
1022         _links = links;
1023         _totals = totals;
1024     }
1025
1026     public void Calculate() => _links.Each(_constants.Any, _constants.Any,
1027         ↪ CalculateCore);
1028
1029     private bool IsElement(ulong link)
1030     {
1031         // _linksInSequence.Contains(link) ||
1032         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1033             ↪ link;
1034     }
1035
1036     private bool CalculateCore(ulong link)
1037     {
1038         // TODO: Проработать защиту от заикливания
1039         // Основано на SequenceWalker.WalkLeft
1040         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1041         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1042         Func<ulong, bool> isElement = IsElement;
1043         void visitLeaf(ulong parent)
1044         {
1045             if (link != parent)
1046             {
1047                 _totals[parent]++;
1048             }
1049         }
1050         void visitNode(ulong parent)
1051         {
1052             if (link != parent)
1053             {
1054                 _totals[parent]++;
1055             }
1056         }
1057         var stack = new Stack();

```

```

1055     var element = link;
1056     if (isElement(element))
1057     {
1058         visitLeaf(element);
1059     }
1060     else
1061     {
1062         while (true)
1063         {
1064             if (isElement(element))
1065             {
1066                 if (stack.Count == 0)
1067                 {
1068                     break;
1069                 }
1070                 element = stack.Pop();
1071                 var source = getSource(element);
1072                 var target = getTarget(element);
1073                 // 06пабoтка элемеHта
1074                 if (isElement(target))
1075                 {
1076                     visitLeaf(target);
1077                 }
1078                 if (isElement(source))
1079                 {
1080                     visitLeaf(source);
1081                 }
1082                 element = source;
1083             }
1084             else
1085             {
1086                 stack.Push(element);
1087                 visitNode(element);
1088                 element = getTarget(element);
1089             }
1090         }
1091         _totals[link]++;
1092         return true;
1093     }
1094 }
1095
1096
1097 private class AllUsagesCollector
1098 {
1099     private readonly ILinks<ulong> _links;
1100     private readonly HashSet<ulong> _usages;
1101
1102     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1103     {
1104         _links = links;
1105         _usages = usages;
1106     }
1107
1108     public bool Collect(ulong link)
1109     {
1110         if (_usages.Add(link))
1111         {
1112             _links.Each(link, _constants.Any, Collect);
1113             _links.Each(_constants.Any, link, Collect);
1114         }
1115         return true;
1116     }
1117 }
1118
1119 private class AllUsagesCollector1
1120 {
1121     private readonly ILinks<ulong> _links;
1122     private readonly HashSet<ulong> _usages;
1123     private readonly ulong _continue;
1124
1125     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1126     {
1127         _links = links;
1128         _usages = usages;
1129         _continue = _links.Constants.Continue;
1130     }
1131
1132     public ulong Collect(ICollection<ulong> link)
1133     {
1134         var linkIndex = _links.GetIndex(link);

```

```

1135         if (_usages.Add(linkIndex))
1136         {
1137             _links.Each(Collect, _constants.Any, linkIndex);
1138         }
1139         return _continue;
1140     }
1141 }
1142
1143 private class AllUsagesCollector2
1144 {
1145     private readonly ILinks<ulong> _links;
1146     private readonly BitString _usages;
1147
1148     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1149     {
1150         _links = links;
1151         _usages = usages;
1152     }
1153
1154     public bool Collect(ulong link)
1155     {
1156         if (_usages.Add((long)link))
1157         {
1158             _links.Each(link, _constants.Any, Collect);
1159             _links.Each(_constants.Any, link, Collect);
1160         }
1161         return true;
1162     }
1163 }
1164
1165 private class AllUsagesIntersectingCollector
1166 {
1167     private readonly SynchronizedLinks<ulong> _links;
1168     private readonly HashSet<ulong> _intersectWith;
1169     private readonly HashSet<ulong> _usages;
1170     private readonly HashSet<ulong> _enter;
1171
1172     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
↪ intersectWith, HashSet<ulong> usages)
1173     {
1174         _links = links;
1175         _intersectWith = intersectWith;
1176         _usages = usages;
1177         _enter = new HashSet<ulong>(); // защита от зацикливания
1178     }
1179
1180     public bool Collect(ulong link)
1181     {
1182         if (_enter.Add(link))
1183         {
1184             if (_intersectWith.Contains(link))
1185             {
1186                 _usages.Add(link);
1187             }
1188             _links.Unsync.Each(link, _constants.Any, Collect);
1189             _links.Unsync.Each(_constants.Any, link, Collect);
1190         }
1191         return true;
1192     }
1193 }
1194
1195 private void CloseInnerConnections(Action<ulong> handler, ulong left, ulong right)
1196 {
1197     TryStepLeftUp(handler, left, right);
1198     TryStepRightUp(handler, right, left);
1199 }
1200
1201 private void AllCloseConnections(Action<ulong> handler, ulong left, ulong right)
1202 {
1203     // Direct
1204     if (left == right)
1205     {
1206         handler(left);
1207     }
1208     var doublet = Links.Unsync.SearchOrDefault(left, right);
1209     if (doublet != _constants.Null)
1210     {
1211         handler(doublet);
1212     }
1213     // Inner

```

```

1214         CloseInnerConnections(handler, left, right);
1215         // Outer
1216         StepLeft(handler, left, right);
1217         StepRight(handler, left, right);
1218         PartialStepRight(handler, left, right);
1219         PartialStepLeft(handler, left, right);
1220     }
1221
1222     private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
1223     ↪ HashSet<ulong> previousMatchings, long startAt)
1224     {
1225         if (startAt >= sequence.Length) // ?
1226         {
1227             return previousMatchings;
1228         }
1229         var secondLinkUsages = new HashSet<ulong>();
1230         AllUsagesCore(sequence[startAt], secondLinkUsages);
1231         secondLinkUsages.Add(sequence[startAt]);
1232         var matchings = new HashSet<ulong>();
1233         //for (var i = 0; i < previousMatchings.Count; i++)
1234         foreach (var secondLinkUsage in secondLinkUsages)
1235         {
1236             foreach (var previousMatching in previousMatchings)
1237             {
1238                 //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1239                 ↪ secondLinkUsage);
1240                 StepRight(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
1241                 TryStepRightUp(matchings.AddAndReturnVoid, secondLinkUsage,
1242                 ↪ previousMatching);
1243                 //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1244                 ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1245                 ↪ желаемым результатам.
1246                 PartialStepRight(matchings.AddAndReturnVoid, previousMatching,
1247                 ↪ secondLinkUsage);
1248             }
1249         }
1250         if (matchings.Count == 0)
1251         {
1252             return matchings;
1253         }
1254         return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1255     }
1256
1257     private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
1258     ↪ links, params ulong[] sequence)
1259     {
1260         if (sequence == null)
1261         {
1262             return;
1263         }
1264         for (var i = 0; i < sequence.Length; i++)
1265         {
1266             if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany &&
1267             ↪ !links.Exists(sequence[i]))
1268             {
1269                 throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1270                 ↪ $"patternSequence[{i}]");
1271             }
1272         }
1273     }
1274
1275     // Pattern Matching -> Key To Triggers
1276     public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1277     {
1278         return Sync.ExecuteReadOperation(() =>
1279         {
1280             patternSequence = Simplify(patternSequence);
1281             if (patternSequence.Length > 0)
1282             {
1283                 EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1284                 var uniqueSequenceElements = new HashSet<ulong>();
1285                 for (var i = 0; i < patternSequence.Length; i++)
1286                 {
1287                     if (patternSequence[i] != _constants.Any && patternSequence[i] !=
1288                     ↪ ZeroOrMany)
1289                     {
1290                         uniqueSequenceElements.Add(patternSequence[i]);
1291                     }
1292                 }
1293             }
1294         });
1295     }

```

```

1281     }
1282 }
1283 var results = new HashSet<ulong>();
1284 foreach (var uniqueSequenceElement in uniqueSequenceElements)
1285 {
1286     AllUsagesCore(uniqueSequenceElement, results);
1287 }
1288 var filteredResults = new HashSet<ulong>();
1289 var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1290 matcher.AddAllPatternMatchedToResults(results);
1291 return filteredResults;
1292 }
1293 return new HashSet<ulong>();
1294 });
1295 }
1296
1297 // Найти все возможные связи между указанным списком связей.
1298 // Находит связи между всеми указанными связями в любом порядке.
1299 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1300 //      ↪ несколько раз в последовательности)
1301 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1302 {
1303     return Sync.ExecuteReadOperation(() =>
1304     {
1305         var results = new HashSet<ulong>();
1306         if (linksToConnect.Length > 0)
1307         {
1308             Links.EnsureEachLinkExists(linksToConnect);
1309             AllUsagesCore(linksToConnect[0], results);
1310             for (var i = 1; i < linksToConnect.Length; i++)
1311             {
1312                 var next = new HashSet<ulong>();
1313                 AllUsagesCore(linksToConnect[i], next);
1314                 results.IntersectWith(next);
1315             }
1316         }
1317         return results;
1318     });
1319 }
1320
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);

```



```

1358         collector.Collect(linksToConnect[i]);
1359         //AllUsagesCore(linksToConnect[i], next);
1360         //results.IntersectWith(next);
1361         results = next;
1362     }
1363 }
1364 return results;
1365 });
1366 }
1367
1368 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1369 {
1370     return Sync.ExecuteReadOperation(() =>
1371     {
1372         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1373         ↪ BitArray((int)_links.Total + 1);
1374         if (linksToConnect.Length > 0)
1375         {
1376             Links.EnsureEachLinkExists(linksToConnect);
1377             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1378             collector1.Collect(linksToConnect[0]);
1379             for (var i = 1; i < linksToConnect.Length; i++)
1380             {
1381                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1382                 ↪ BitArray((int)_links.Total + 1);
1383                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1384                 collector.Collect(linksToConnect[i]);
1385                 results = results.And(next);
1386             }
1387             return results.GetSetUInt64Indices();
1388         }
1389     });
1390 }
1391
1392 private static ulong[] Simplify(ulong[] sequence)
1393 {
1394     // Считаем новый размер последовательности
1395     long newLength = 0;
1396     var zeroOrManyStepped = false;
1397     for (var i = 0; i < sequence.Length; i++)
1398     {
1399         if (sequence[i] == ZeroOrMany)
1400         {
1401             if (zeroOrManyStepped)
1402             {
1403                 continue;
1404             }
1405             zeroOrManyStepped = true;
1406         }
1407         else
1408         {
1409             //if (zeroOrManyStepped) Is it efficient?
1410             zeroOrManyStepped = false;
1411             newLength++;
1412         }
1413     }
1414     // Строим новую последовательность
1415     zeroOrManyStepped = false;
1416     var newSequence = new ulong[newLength];
1417     long j = 0;
1418     for (var i = 0; i < sequence.Length; i++)
1419     {
1420         //var current = zeroOrManyStepped;
1421         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1422         //if (current && zeroOrManyStepped)
1423         //    continue;
1424         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1425         //if (zeroOrManyStepped && newZeroOrManyStepped)
1426         //    continue;
1427         //zeroOrManyStepped = newZeroOrManyStepped;
1428         if (sequence[i] == ZeroOrMany)
1429         {
1430             if (zeroOrManyStepped)
1431             {
1432                 continue;
1433             }
1434             zeroOrManyStepped = true;
1435         }
1436         else

```

```

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

```

```

1509     }
1510     else
1511     {
1512         var nextRightLink = middleLinks[rightBound];
1513         var elements = GetLeftElements(rightLink, nextRightLink);
1514         if (leftBound <= rightBound)
1515         {
1516             for (var i = elements.Length - 1; i >= 0; i--)
1517             {
1518                 var element = elements[i];
1519                 if (element != 0)
1520                 {
1521                     CollectMatchingSequences(leftLink, leftBound, middleLinks,
1522                                             ↪ elements[i], rightBound - 1, ref results);
1523                 }
1524             }
1525         }
1526         else
1527         {
1528             for (var i = elements.Length - 1; i >= 0; i--)
1529             {
1530                 var element = elements[i];
1531                 if (element != 0)
1532                 {
1533                     results.Add(element);
1534                 }
1535             }
1536         }
1537     }
1538 }
1539 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1540 {
1541     var result = new ulong[5];
1542     TryStepRight(startLink, rightLink, result, 0);
1543     Links.Each(_constants.Any, startLink, couple =>
1544     {
1545         if (couple != startLink)
1546         {
1547             if (TryStepRight(couple, rightLink, result, 2))
1548             {
1549                 return false;
1550             }
1551         }
1552         return true;
1553     });
1554     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1555     {
1556         result[4] = startLink;
1557     }
1558     return result;
1559 }
1560
1561 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1562 {
1563     var added = 0;
1564     Links.Each(startLink, _constants.Any, couple =>
1565     {
1566         if (couple != startLink)
1567         {
1568             var coupleTarget = Links.GetTarget(couple);
1569             if (coupleTarget == rightLink)
1570             {
1571                 result[offset] = couple;
1572                 if (++added == 2)
1573                 {
1574                     return false;
1575                 }
1576             }
1577             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1578                 ↪ == Net.And &&
1579             {
1580                 result[offset + 1] = couple;
1581                 if (++added == 2)
1582                 {
1583                     return false;
1584                 }
1585             }
1586         }
1587     }

```

```

1585     }
1586     return true;
1587 });
1588 return added > 0;
1589 }
1590
1591 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1592 {
1593     var result = new ulong[5];
1594     TryStepLeft(startLink, leftLink, result, 0);
1595     Links.Each(startLink, _constants.Any, couple =>
1596     {
1597         if (couple != startLink)
1598         {
1599             if (TryStepLeft(couple, leftLink, result, 2))
1600             {
1601                 return false;
1602             }
1603         }
1604         return true;
1605     });
1606     if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1607     {
1608         result[4] = leftLink;
1609     }
1610     return result;
1611 }
1612
1613 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1614 {
1615     var added = 0;
1616     Links.Each(_constants.Any, startLink, couple =>
1617     {
1618         if (couple != startLink)
1619         {
1620             var coupleSource = Links.GetSource(couple);
1621             if (coupleSource == leftLink)
1622             {
1623                 result[offset] = couple;
1624                 if (++added == 2)
1625                 {
1626                     return false;
1627                 }
1628             }
1629             else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1630                 ↪ == Net.And &&
1631             {
1632                 result[offset + 1] = couple;
1633                 if (++added == 2)
1634                 {
1635                     return false;
1636                 }
1637             }
1638         }
1639         return true;
1640     });
1641     return added > 0;
1642 }
1643
1644 #endregion
1645
1646 #region Walkers
1647
1648 public class PatternMatcher : RightSequenceWalker<ulong>
1649 {
1650     private readonly Sequences _sequences;
1651     private readonly ulong[] _patternSequence;
1652     private readonly HashSet<LinkIndex> _linksInSequence;
1653     private readonly HashSet<LinkIndex> _results;
1654
1655     #region Pattern Match
1656
1657     enum PatternBlockType
1658     {
1659         Undefined,
1660         Gap,
1661         Elements
1662     }
1663
1664     struct PatternBlock

```

```

1664 {
1665     public PatternBlockType Type;
1666     public long Start;
1667     public long Stop;
1668 }
1669
1670 private readonly List<PatternBlock> _pattern;
1671 private int _patternPosition;
1672 private long _sequencePosition;
1673
1674 #endregion
1675
1676 public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1677     ↳ HashSet<LinkIndex> results)
1678     : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1679 {
1680     _sequences = sequences;
1681     _patternSequence = patternSequence;
1682     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1683     ↳ _constants.Any && x != ZeroOrMany));
1684     _results = results;
1685     _pattern = CreateDetailedPattern();
1686 }
1687
1688 protected override bool IsElement(IList<ulong> link) =>
1689     ↳ _linksInSequence.Contains(Links.GetIndex(link)) || base.IsElement(link);
1690
1691 public bool PatternMatch(LinkIndex sequenceToMatch)
1692 {
1693     _patternPosition = 0;
1694     _sequencePosition = 0;
1695     foreach (var part in Walk(sequenceToMatch))
1696     {
1697         if (!PatternMatchCore(Links.GetIndex(part)))
1698         {
1699             break;
1700         }
1701     }
1702     return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1703     ↳ - 1 && _pattern[_patternPosition].Start == 0);
1704 }
1705
1706 private List<PatternBlock> CreateDetailedPattern()
1707 {
1708     var pattern = new List<PatternBlock>();
1709     var patternBlock = new PatternBlock();
1710     for (var i = 0; i < _patternSequence.Length; i++)
1711     {
1712         if (patternBlock.Type == PatternBlockType.Undefined)
1713         {
1714             if (_patternSequence[i] == _constants.Any)
1715             {
1716                 patternBlock.Type = PatternBlockType.Gap;
1717                 patternBlock.Start = 1;
1718                 patternBlock.Stop = 1;
1719             }
1720             else if (_patternSequence[i] == ZeroOrMany)
1721             {
1722                 patternBlock.Type = PatternBlockType.Gap;
1723                 patternBlock.Start = 0;
1724                 patternBlock.Stop = long.MaxValue;
1725             }
1726             else
1727             {
1728                 patternBlock.Type = PatternBlockType.Elements;
1729                 patternBlock.Start = i;
1730                 patternBlock.Stop = i;
1731             }
1732         }
1733         else if (patternBlock.Type == PatternBlockType.Elements)
1734         {
1735             if (_patternSequence[i] == _constants.Any)
1736             {
1737                 pattern.Add(patternBlock);
1738                 patternBlock = new PatternBlock
1739                 {
1740                     Type = PatternBlockType.Gap,
1741                     Start = 1,
1742                     Stop = 1
1743                 };
1744             }
1745         }
1746     }
1747 }

```

```

1740     }
1741     else if (_patternSequence[i] == ZeroOrMany)
1742     {
1743         pattern.Add(patternBlock);
1744         patternBlock = new PatternBlock
1745         {
1746             Type = PatternBlockType.Gap,
1747             Start = 0,
1748             Stop = long.MaxValue
1749         };
1750     }
1751     else
1752     {
1753         patternBlock.Stop = i;
1754     }
1755 }
1756 else // patternBlock.Type == PatternBlockType.Gap
1757 {
1758     if (_patternSequence[i] == _constants.Any)
1759     {
1760         patternBlock.Start++;
1761         if (patternBlock.Stop < patternBlock.Start)
1762         {
1763             patternBlock.Stop = patternBlock.Start;
1764         }
1765     }
1766     else if (_patternSequence[i] == ZeroOrMany)
1767     {
1768         patternBlock.Stop = long.MaxValue;
1769     }
1770     else
1771     {
1772         pattern.Add(patternBlock);
1773         patternBlock = new PatternBlock
1774         {
1775             Type = PatternBlockType.Elements,
1776             Start = i,
1777             Stop = i
1778         };
1779     }
1780 }
1781 }
1782 if (patternBlock.Type != PatternBlockType.Undefined)
1783 {
1784     pattern.Add(patternBlock);
1785 }
1786 return pattern;
1787 }
1788
1789 /** match: search for regexp anywhere in text */
1790 /**int match(char* regexp, char* text)
1791 /**{
1792 /**    do
1793 /**    {
1794 /**    } while (*text++ != '\0');
1795 /**    return 0;
1796 /**}
1797
1798 /** matchhere: search for regexp at beginning of text */
1799 /**int matchhere(char* regexp, char* text)
1800 /**{
1801 /**    if (regexp[0] == '\0')
1802 /**        return 1;
1803 /**    if (regexp[1] == '*')
1804 /**        return matchstar(regexp[0], regexp + 2, text);
1805 /**    if (regexp[0] == '$' && regexp[1] == '\0')
1806 /**        return *text == '\0';
1807 /**    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1808 /**        return matchhere(regexp + 1, text + 1);
1809 /**    return 0;
1810 /**}
1811
1812 /** matchstar: search for c*regexp at beginning of text */
1813 /**int matchstar(int c, char* regexp, char* text)
1814 /**{
1815 /**    do
1816 /**    {        /* a * matches zero or more instances */
1817 /**        if (matchhere(regexp, text))
1818 /**            return 1;

```

```

1819 // } while (*text != '\0' && (*text++ == c || c == '.'));
1820 // return 0;
1821 //}
1822
1823 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1824 ↪ long maximumGap)
1825 //{
1826 //    mininumGap = 0;
1827 //    maximumGap = 0;
1828 //    element = 0;
1829 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1830 //    {
1831 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1832 //            mininumGap++;
1833 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1834 //            maximumGap = long.MaxValue;
1835 //        else
1836 //            break;
1837 //    }
1838 //    if (maximumGap < mininumGap)
1839 //        maximumGap = mininumGap;
1840 //}
1841
1842 private bool PatternMatchCore(LinkIndex element)
1843 {
1844     if (_patternPosition >= _pattern.Count)
1845     {
1846         _patternPosition = -2;
1847         return false;
1848     }
1849     var currentPatternBlock = _pattern[_patternPosition];
1850     if (currentPatternBlock.Type == PatternBlockType.Gap)
1851     {
1852         //var currentMatchingBlockLength = (_sequencePosition -
1853 ↪ _lastMatchedBlockPosition);
1854         if (_sequencePosition < currentPatternBlock.Start)
1855         {
1856             _sequencePosition++;
1857             return true; // Двигаемся дальше
1858         }
1859         // Это последний блок
1860         if (_pattern.Count == _patternPosition + 1)
1861         {
1862             _patternPosition++;
1863             _sequencePosition = 0;
1864             return false; // Полное соответствие
1865         }
1866         else
1867         {
1868             if (_sequencePosition > currentPatternBlock.Stop)
1869             {
1870                 return false; // Соответствие невозможно
1871             }
1872             var nextPatternBlock = _pattern[_patternPosition + 1];
1873             if (_patternSequence[nextPatternBlock.Start] == element)
1874             {
1875                 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1876                 {
1877                     _patternPosition++;
1878                     _sequencePosition = 1;
1879                 }
1880                 else
1881                 {
1882                     _patternPosition += 2;
1883                     _sequencePosition = 0;
1884                 }
1885             }
1886         }
1887     }
1888     else // currentPatternBlock.Type == PatternBlockType.Elements
1889     {
1890         var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1891         if (_patternSequence[patternElementPosition] != element)
1892         {
1893             return false; // Соответствие невозможно
1894         }
1895         if (patternElementPosition == currentPatternBlock.Stop)
1896         {

```

```

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

```

./Platform.Data.Doublets/Sequences/Sequences.Experiments.ReadSequence.cs

```

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

```



```

18         if (isElement(sequence))
19         {
20             return array;
21         }
22     }
23
24     bool hasElements;
25     do
26     {
27         length *= 2;
28     #if USEARRAYPOOL
29         var nextArray = ArrayPool.Allocate<ulong>(length);
30     #else
31         var nextArray = new ulong[length];
32     #endif
33     hasElements = false;
34     for (var i = 0; i < array.Length; i++)
35     {
36         var candidate = array[i];
37         if (candidate == 0)
38         {
39             continue;
40         }
41         var doubletOffset = i * 2;
42         if (isElement(candidate))
43         {
44             nextArray[doubletOffset] = candidate;
45         }
46         else
47         {
48             var link = links.GetLink(candidate);
49             var linkSource = links.GetSource(link);
50             var linkTarget = links.GetTarget(link);
51             nextArray[doubletOffset] = linkSource;
52             nextArray[doubletOffset + 1] = linkTarget;
53             if (!hasElements)
54             {
55                 hasElements = !(isElement(linkSource) && isElement(linkTarget));
56             }
57         }
58     }
59     #if USEARRAYPOOL
60     if (array.Length > 1)
61     {
62         ArrayPool.Free(array);
63     }
64     #endif
65     array = nextArray;
66 }
67 while (hasElements);
68 var filledElementsCount = CountFilledElements(array);
69 if (filledElementsCount == array.Length)
70 {
71     return array;
72 }
73 else
74 {
75     return CopyFilledElements(array, filledElementsCount);
76 }
77 }
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 private static ulong[] CopyFilledElements(ulong[] array, int filledElementsCount)
81 {
82     var finalArray = new ulong[filledElementsCount];
83     for (int i = 0, j = 0; i < array.Length; i++)
84     {
85         if (array[i] > 0)
86         {
87             finalArray[j] = array[i];
88             j++;
89         }
90     }
91     #if USEARRAYPOOL
92     ArrayPool.Free(array);
93     #endif
94     return finalArray;
95 }
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

./Platform.Data.Doublets/Sequences/SequencesExtensions.cs

```

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

```

./Platform.Data.Doublets/Sequences/SequencesIndexer.cs

```

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

```

```

39 public bool BulkIndex(TLink[] sequence)
40 {
41     var indexed = true;
42     var i = sequence.Length;
43     var links = _links.Unsync;
44     _links.SyncRoot.ExecuteReadOperation(() =>
45     {
46         while (--i >= 1 && (indexed =
47             ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
48             ↪ sequence[i]), _null))) { }
49     });
50     if (indexed == false)
51     {
52         _links.SyncRoot.ExecuteWriteOperation(() =>
53         {
54             for (; i >= 1; i--)
55             {
56                 links.GetOrCreate(sequence[i - 1], sequence[i]);
57             }
58         });
59     }
60     return indexed;
61 }
62
63 public bool BulkIndexUnsync(TLink[] sequence)
64 {
65     var indexed = true;
66     var i = sequence.Length;
67     var links = _links.Unsync;
68     while (--i >= 1 && (indexed =
69         ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1], sequence[i]),
70         ↪ _null))) { }
71     for (; i >= 1; i--)
72     {
73         links.GetOrCreate(sequence[i - 1], sequence[i]);
74     }
75     return indexed;
76 }
77
78 public bool CheckIndex(ICollection<TLink> sequence)
79 {
80     var indexed = true;
81     var i = sequence.Count;
82     while (--i >= 1 && (indexed =
83         ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
84         ↪ _null))) { }
85     return indexed;
86 }
87 }
88 }

```

./Platform.Data.Doublets/Sequences/SequencesOptions.cs

```

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

```

```

26 public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
27 public SequencesIndexer<TLink> Indexer { get; set; }
28
29 // TODO: Реализовать компактификацию при чтении
30 //public bool EnforceSingleSequenceVersionOnRead { get; set; }
31 //public bool UseRequestMarker { get; set; }
32 //public bool StoreRequestResults { get; set; }
33
34 public void InitOptions(ISynchronizedLinks<TLink> links)
35 {
36     if (UseSequenceMarker)
37     {
38         if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
39         {
40             SequenceMarkerLink = links.CreatePoint();
41         }
42         else
43         {
44             if (!links.Exists(SequenceMarkerLink))
45             {
46                 var link = links.CreatePoint();
47                 if (!_equalityComparer.Equals(link, SequenceMarkerLink))
48                 {
49                     throw new InvalidOperationException("Cannot recreate sequence marker
50                     ↪ link.");
51                 }
52             }
53             if (MarkedSequenceMatcher == null)
54             {
55                 MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
56                 ↪ SequenceMarkerLink);
57             }
58             var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
59             if (UseCompression)
60             {
61                 if (LinksToSequenceConverter == null)
62                 {
63                     ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
64                     if (UseSequenceMarker)
65                     {
66                         totalSequenceSymbolFrequencyCounter = new
67                         ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
68                         ↪ MarkedSequenceMatcher);
69                     }
70                     else
71                     {
72                         totalSequenceSymbolFrequencyCounter = new
73                         ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
74                     }
75                     var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
76                     ↪ totalSequenceSymbolFrequencyCounter);
77                     var compressingConverter = new CompressingConverter<TLink>(links,
78                     ↪ balancedVariantConverter, doubletFrequenciesCache);
79                     LinksToSequenceConverter = compressingConverter;
80                 }
81             }
82             else
83             {
84                 if (LinksToSequenceConverter == null)
85                 {
86                     LinksToSequenceConverter = balancedVariantConverter;
87                 }
88             }
89             if (UseIndex && Indexer == null)
90             {
91                 Indexer = new SequencesIndexer<TLink>(links);
92             }
93         }
94     }
95
96 public void ValidateOptions()
97 {
98     if (UseGarbageCollection && !UseSequenceMarker)
99     {
100         throw new NotSupportedException("To use garbage collection UseSequenceMarker
101         ↪ option must be on.");
102     }
103 }

```

```

96     }
97 }
98 }

```

./Platform.Data.Doublets/Sequences/UnicodeMap.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using Platform.Data.Sequences;
7
8  namespace Platform.Data.Doublets.Sequences
9  {
10     public class UnicodeMap
11     {
12         public static readonly ulong FirstCharLink = 1;
13         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
14         public static readonly ulong MapSize = 1 + char.MaxValue;
15
16         private readonly ILinks<ulong> _links;
17         private bool _initialized;
18
19         public UnicodeMap(ILinks<ulong> links) => _links = links;
20
21         public static UnicodeMap InitNew(ILinks<ulong> links)
22         {
23             var map = new UnicodeMap(links);
24             map.Init();
25             return map;
26         }
27
28         public void Init()
29         {
30             if (_initialized)
31             {
32                 return;
33             }
34             _initialized = true;
35             var firstLink = _links.CreatePoint();
36             if (firstLink != FirstCharLink)
37             {
38                 _links.Delete(firstLink);
39             }
40             else
41             {
42                 for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
43                 {
44                     // From NIL to It (NIL -> Character) transformation meaning, (or infinite
45                     // ↪ amount of NIL characters before actual Character)
46                     var createdLink = _links.CreatePoint();
47                     _links.Update(createdLink, firstLink, createdLink);
48                     if (createdLink != i)
49                     {
50                         throw new InvalidOperationException("Unable to initialize UTF 16
51                         ↪ table.");
52                     }
53                 }
54             }
55         }
56
57         // 0 - null link
58         // 1 - nil character (0 character)
59         // ...
60         // 65536 (0(1) + 65535 = 65536 possible values)
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public static ulong FromCharToLink(char character) => (ulong)character + 1;
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         public static char FromLinkToChar(ulong link) => (char)(link - 1);
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         public static bool IsCharLink(ulong link) => link <= MapSize;
70
71         public static string FromLinksToString(IList<ulong> linksList)
72         {
73             var sb = new StringBuilder();
74             for (int i = 0; i < linksList.Count; i++)
75             {

```

```

74         sb.Append(FromLinkToChar(linksList[i]));
75     }
76     return sb.ToString();
77 }
78
79 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
80 {
81     var sb = new StringBuilder();
82     if (links.Exists(link))
83     {
84         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
85             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
86             element =>
87             {
88                 sb.Append(FromLinkToChar(element));
89                 return true;
90             }
91         );
92     }
93     return sb.ToString();
94 }
95
96 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
97     ↪ chars.Length);
98
99 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
100 {
101     // char array to ulong array
102     var linksSequence = new ulong[count];
103     for (var i = 0; i < count; i++)
104     {
105         linksSequence[i] = FromCharToLink(chars[i]);
106     }
107     return linksSequence;
108 }
109
110 public static ulong[] FromStringToLinkArray(string sequence)
111 {
112     // char array to ulong array
113     var linksSequence = new ulong[sequence.Length];
114     for (var i = 0; i < sequence.Length; i++)
115     {
116         linksSequence[i] = FromCharToLink(sequence[i]);
117     }
118     return linksSequence;
119 }
120
121 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
122 {
123     var result = new List<ulong[]>();
124     var offset = 0;
125     while (offset < sequence.Length)
126     {
127         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
128         var relativeLength = 1;
129         var absoluteLength = offset + relativeLength;
130         while (absoluteLength < sequence.Length &&
131             currentCategory ==
132             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
133         {
134             relativeLength++;
135             absoluteLength++;
136         }
137         // char array to ulong array
138         var innerSequence = new ulong[relativeLength];
139         var maxLength = offset + relativeLength;
140         for (var i = offset; i < maxLength; i++)
141         {
142             innerSequence[i - offset] = FromCharToLink(sequence[i]);
143         }
144         result.Add(innerSequence);
145         offset += relativeLength;
146     }
147     return result;
148 }
149
150 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
151 {
152     var result = new List<ulong[]>();
153     var offset = 0;

```

```

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

```

./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Collections.Stacks;
4
5 namespace Platform.Data.Doublets.Sequences.Walkers
6 {
7     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
8     {
9         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack)
10         ↳ { }
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         protected override TLink GetNextElementAfterPop(TLink element) =>
14         ↳ Links.GetSource(element);
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected override TLink GetNextElementAfterPush(TLink element) =>
18         ↳ Links.GetTarget(element);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override IEnumerable<IList<TLink>> WalkContents(IList<TLink> 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 }

```

./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs

```

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

```

```

3  using Platform.Collections.Stacks;
4
5  namespace Platform.Data.Doublets.Sequences.Walkers
6  {
7      public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
8      {
9          public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
10             ↪ stack) { }
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          protected override TLink GetNextElementAfterPop(TLink element) =>
14             ↪ Links.GetTarget(element);
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override TLink GetNextElementAfterPush(TLink element) =>
18             ↪ Links.GetSource(element);
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override IEnumerable<IList<TLink>> WalkContents(IList<TLink> 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  }

```

./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Stacks;
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         private readonly IStack<TLink> _stack;
12
13         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : base(links) =>
14             ↪ _stack = stack;
15
16         public IEnumerable<IList<TLink>> Walk(TLink sequence)
17         {
18             _stack.Clear();
19             var element = sequence;
20             var elementValues = Links.GetLink(element);
21             if (IsElement(elementValues))
22             {
23                 yield return elementValues;
24             }
25             else
26             {
27                 while (true)
28                 {
29                     if (IsElement(elementValues))
30                     {
31                         if (_stack.IsEmpty)
32                         {
33                             break;
34                         }
35                         element = _stack.Pop();
36                         elementValues = Links.GetLink(element);
37                         foreach (var output in WalkContents(elementValues))
38                         {
39                             yield return output;
40                         }
41                         element = GetNextElementAfterPop(element);
42                         elementValues = Links.GetLink(element);
43                     }
44                     else
45                     {
46                         _stack.Push(element);
47                     }
48                 }
49             }
50         }
51     }

```



```

45         element = GetNextElementAfterPush(element);
46         elementValues = Links.GetLink(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 TLink GetNextElementAfterPop(TLink element);
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected abstract TLink GetNextElementAfterPush(TLink element);
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected abstract IEnumerable<IList<TLink>> WalkContents(IList<TLink> element);
64 }

```

./Platform.Data.Doublets/Stacks/Stack.cs

```

1  using System.Collections.Generic;
2  using Platform.Collections.Stacks;
3
4  namespace Platform.Data.Doublets.Stacks
5  {
6      public class Stack<TLink> : IStack<TLink>
7      {
8          private static readonly EqualityComparer<TLink> _equalityComparer =
9              ⇨ EqualityComparer<TLink>.Default;
10
11          private readonly ILinks<TLink> _links;
12          private readonly TLink _stack;
13
14          public bool IsEmpty => _equalityComparer.Equals(Peek(), _stack);
15
16          public Stack(ILinks<TLink> links, TLink stack)
17          {
18              _links = links;
19              _stack = stack;
20          }
21
22          private TLink GetStackMarker() => _links.GetSource(_stack);
23
24          private TLink GetTop() => _links.GetTarget(_stack);
25
26          public TLink Peek() => _links.GetTarget(GetTop());
27
28          public TLink Pop()
29          {
30              var element = Peek();
31              if (!_equalityComparer.Equals(element, _stack))
32              {
33                  var top = GetTop();
34                  var previousTop = _links.GetSource(top);
35                  _links.Update(_stack, GetStackMarker(), previousTop);
36                  _links.Delete(top);
37              }
38              return element;
39          }
40
41          public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
42              ⇨ _links.GetOrCreate(GetTop(), element));
43      }
44 }

```

./Platform.Data.Doublets/Stacks/StackExtensions.cs

```

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

```

```

11     }
12 }

./Platform.Data.Doublets/SynchronizedLinks.cs
1  using System;
2  using System.Collections.Generic;
3  using Platform.Data.Constants;
4  using Platform.Data.Doublets;
5  using Platform.Threading.Synchronization;
6
7  namespace Platform.Data.Doublets
8  {
9      /// <remarks>
10     /// TODO: Autogeneration of synchronized wrapper (decorator).
11     /// TODO: Try to unfold code of each method using IL generation for performance improvements.
12     /// TODO: Or even to unfold multiple layers of implementations.
13     /// </remarks>
14     public class SynchronizedLinks<T> : ISynchronizedLinks<T>
15     {
16         public LinksCombinedConstants<T, T, int> Constants { get; }
17         public ISynchronization SyncRoot { get; }
18         public ILinks<T> Sync { get; }
19         public ILinks<T> Unsync { get; }
20
21         public SynchronizedLinks(ILinks<T> links) : this(new ReaderWriterLockSynchronization(),
22             ↪ links) { }
23
24         public SynchronizedLinks(ISynchronization synchronization, ILinks<T> links)
25         {
26             SyncRoot = synchronization;
27             Sync = this;
28             Unsync = links;
29             Constants = links.Constants;
30         }
31
32         public T Count(IList<T> restriction) => SyncRoot.ExecuteReadOperation(restriction,
33             ↪ Unsync.Count);
34         public T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
35             ↪ SyncRoot.ExecuteReadOperation(handler, restrictions, (handler1, restrictions1) =>
36             ↪ Unsync.Each(handler1, restrictions1));
37         public T Create() => SyncRoot.ExecuteWriteOperation(Unsync.Create);
38         public T Update(IList<T> restrictions) => SyncRoot.ExecuteWriteOperation(restrictions,
39             ↪ Unsync.Update);
40         public void Delete(T link) => SyncRoot.ExecuteWriteOperation(link, Unsync.Delete);
41
42         //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
43         //    ↪ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
44         //{
45         //    if (restriction != null && substitution != null &&
46         //        !substitution.EqualTo(restriction))
47         //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
48         //            ↪ substitution, substitutedHandler, Unsync.Trigger);
49         //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
50         //        ↪ substitutedHandler, Unsync.Trigger);
51         //}
52     }
53 }

```

```

./Platform.Data.Doublets/UInt64Link.cs
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Platform.Exceptions;
5  using Platform.Ranges;
6  using Platform.Singletons;
7  using Platform.Collections.Lists;
8  using Platform.Data.Constants;
9
10 namespace Platform.Data.Doublets
11 {
12     /// <summary>
13     /// Структура описывающая уникальную связь.
14     /// </summary>
15     public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
16     {
17         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
18             ↪ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
19     }
20 }

```

```

19 private const int Length = 3;
20
21 public readonly ulong Index;
22 public readonly ulong Source;
23 public readonly ulong Target;
24
25 public static readonly UInt64Link Null = new UInt64Link();
26
27 public UInt64Link(params ulong[] values)
28 {
29     Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
        ↳ _constants.Null;
30     Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
        ↳ _constants.Null;
31     Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
        ↳ _constants.Null;
32 }
33
34 public UInt64Link(ICollection<ulong> values)
35 {
36     Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
        ↳ _constants.Null;
37     Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
        ↳ _constants.Null;
38     Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
        ↳ _constants.Null;
39 }
40
41 public UInt64Link(ulong index, ulong source, ulong target)
42 {
43     Index = index;
44     Source = source;
45     Target = target;
46 }
47
48 public UInt64Link(ulong source, ulong target)
49 : this(_constants.Null, source, target)
50 {
51     Source = source;
52     Target = target;
53 }
54
55 public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
    ↳ target);
56
57 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
58
59 public bool IsNull() => Index == _constants.Null
60     && Source == _constants.Null
61     && Target == _constants.Null;
62
63 public override bool Equals(object other) => other is UInt64Link &&
    ↳ Equals((UInt64Link)other);
64
65 public bool Equals(UInt64Link other) => Index == other.Index
66     && Source == other.Source
67     && Target == other.Target;
68
69 public static string ToString(ulong index, ulong source, ulong target) => $"{index}:
    ↳ {source}->{target}";
70
71 public static string ToString(ulong source, ulong target) => $"{source}->{target}";
72
73 public static implicit operator ulong[] (UInt64Link link) => link.ToArray();
74
75 public static implicit operator UInt64Link(ulong[] linkArray) => new
    ↳ UInt64Link(linkArray);
76
77 public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
    ↳ : ToString(Index, Source, Target);
78
79 #region IList
80
81 public ulong this[int index]
82 {
83     get
84     {
85         Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
            ↳ nameof(index));
86         if (index == _constants.IndexPart)

```

```

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

```

./Platform.Data.Doublets/UInt64LinkExtensions.cs

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

./Platform.Data.Doublets/UInt64LinksExtensions.cs

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

```

64 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
    ↳ Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
65 {
66     var sb = new StringBuilder();
67     var visited = new HashSet<ulong>();
68     links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
    ↳ innerSb.Append(link.Index), renderIndex, renderDebug);
69     return sb.ToString();
70 }
71
72 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
    ↳ Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
    ↳ bool renderIndex = false, bool renderDebug = false)
73 {
74     var sb = new StringBuilder();
75     var visited = new HashSet<ulong>();
76     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
    ↳ renderDebug);
77     return sb.ToString();
78 }
79
80 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
    ↳ HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
    ↳ Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false, bool
    ↳ renderDebug = false)
81 {
82     if (sb == null)
83     {
84         throw new ArgumentNullException(nameof(sb));
85     }
86     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
    ↳ Constants.Itself)
87     {
88         return;
89     }
90     if (links.Exists(linkIndex))
91     {
92         if (visited.Add(linkIndex))
93         {
94             sb.Append('(');
95             var link = new UInt64Link(links.GetLink(linkIndex));
96             if (renderIndex)
97             {
98                 sb.Append(link.Index);
99                 sb.Append(':');
100             }
101             if (link.Source == link.Index)
102             {
103                 sb.Append(link.Index);
104             }
105             else
106             {
107                 var source = new UInt64Link(links.GetLink(link.Source));
108                 if (isElement(source))
109                 {
110                     appendElement(sb, source);
111                 }
112                 else
113                 {
114                     links.AppendStructure(sb, visited, source.Index, isElement,
    ↳ appendElement, renderIndex);
115                 }
116             }
117             sb.Append(' ');
118             if (link.Target == link.Index)
119             {
120                 sb.Append(link.Index);
121             }
122             else
123             {
124                 var target = new UInt64Link(links.GetLink(link.Target));
125                 if (isElement(target))
126                 {
127                     appendElement(sb, target);
128                 }
129                 else
130                 {

```

```

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

```

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```

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

```

```

52     /// }
53     ///
54     /// public UniqueTimestamp Timestamp
55     /// {
56     ///     get
57     ///     {
58     ///         return (UniqueTimestamp)mask & TransactionIdCombined;
59     ///     }
60     /// }
61     ///
62     /// public TransactionItemType Type
63     /// {
64     ///     get
65     ///     {
66     ///         // Использовать по одному биту из TransactionId и Timestamp,
67     ///         // для значения в 2 бита, которое представляет тип операции
68     ///         throw new NotImplementedException();
69     ///     }
70     /// }
71     /// }
72     ///
73     /// private struct Transition
74     /// {
75     ///     public TransitionHeader Header;
76     ///     public Link Source;
77     ///     public Link Linker;
78     ///     public Link Target;
79     /// }
80     ///
81     /// </remarks>
82     public struct Transition
83     {
84         public static readonly long Size = Structure<Transition>.Size;
85
86         public readonly ulong TransactionId;
87         public readonly UInt64Link Before;
88         public readonly UInt64Link After;
89         public readonly Timestamp Timestamp;
90
91         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
92             ↪ transactionId, UInt64Link before, UInt64Link after)
93         {
94             TransactionId = transactionId;
95             Before = before;
96             After = after;
97             Timestamp = uniqueTimestampFactory.Create();
98         }
99
100        public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
101            ↪ transactionId, UInt64Link before)
102            : this(uniqueTimestampFactory, transactionId, before, default)
103        {
104        }
105
106        public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
107            : this(uniqueTimestampFactory, transactionId, default, default)
108        {
109        }
110
111        public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
112            ↪ {After}";
113    }
114
115    /// <remarks>
116    /// Другие варианты реализации транзакций (атомарности):
117    /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
118    ///     ↪ Target)) и индексов.
119    /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
120    ///     ↪ потребуется решить вопрос
121    ///     со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
122    ///     ↪ пересечениями идентификаторов.
123    ///
124    /// Где хранить промежуточный список транзакций?
125    ///
126    /// В оперативной памяти:
127    /// Минусы:
128    /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
129    ///     так как нужно отдельно выделять память под список трансформаций.

```



```

124 ///      2. Выделенной оперативной памяти может не хватить, в том случае,
125 ///      если транзакция использует слишком много трансформаций.
126 ///      -> Можно использовать жёсткий диск для слишком длинных транзакций.
127 ///      -> Максимальный размер списка трансформаций можно ограничить / задать
    ↪ константой.
128 ///      3. При подтверждении транзакции (Commit) все трансформации записываются разом
    ↪ создавая задержку.
129 ///
130 /// На жёстком диске:
131 /// Минусы:
132 /// 1. Длительный отклик, на запись каждой трансформации.
133 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
134 /// -> Это может решаться упаковкой/исключением дублирующих операций.
135 /// -> Также это может решаться тем, что короткие транзакции вообще
136 ///     не будут записываться в случае отката.
137 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
    ↪ операции (трансформации)
    ↪ будут записаны в лог.
138 ///
139 ///
140 /// </remarks>
141 public class Transaction : DisposableBase
142 {
143     private readonly Queue<Transition> _transitions;
144     private readonly UInt64LinksTransactionsLayer _layer;
145     public bool IsCommitted { get; private set; }
146     public bool IsReverted { get; private set; }
147
148     public Transaction(UInt64LinksTransactionsLayer layer)
149     {
150         _layer = layer;
151         if (_layer._currentTransactionId != 0)
152         {
153             throw new NotSupportedException("Nested transactions not supported.");
154         }
155         IsCommitted = false;
156         IsReverted = false;
157         _transitions = new Queue<Transition>();
158         SetCurrentTransaction(layer, this);
159     }
160
161     public void Commit()
162     {
163         EnsureTransactionAllowsWriteOperations(this);
164         while (_transitions.Count > 0)
165         {
166             var transition = _transitions.Dequeue();
167             _layer._transitions.Enqueue(transition);
168         }
169         _layer._lastCommittedTransactionId = _layer._currentTransactionId;
170         IsCommitted = true;
171     }
172
173     private void Revert()
174     {
175         EnsureTransactionAllowsWriteOperations(this);
176         var transitionsToRevert = new Transition[_transitions.Count];
177         _transitions.CopyTo(transitionsToRevert, 0);
178         for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
179         {
180             _layer.RevertTransition(transitionsToRevert[i]);
181         }
182         IsReverted = true;
183     }
184
185     public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
    ↪ Transaction transaction)
186     {
187         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
188         layer._currentTransactionTransitions = transaction._transitions;
189         layer._currentTransaction = transaction;
190     }
191
192     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
193     {
194         if (transaction.IsReverted)
195         {
196             throw new InvalidOperationException("Transation is reverted.");
197         }
198         if (transaction.IsCommitted)

```

```

199         {
200             throw new InvalidOperationException("Transation is committed.");
201         }
202     }
203
204     protected override void Dispose(bool manual, bool wasDisposed)
205     {
206         if (!wasDisposed && _layer != null && !_layer.IsDisposed)
207         {
208             if (!IsCommitted && !IsReverted)
209             {
210                 Revert();
211             }
212             _layer.ResetCurrentTransation();
213         }
214     }
215 }
216
217 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
218
219 private readonly string _logAddress;
220 private readonly FileStream _log;
221 private readonly Queue<Transition> _transitions;
222 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
223 private Task _transitionsPusher;
224 private Transition _lastCommittedTransition;
225 private ulong _currentTransactionId;
226 private Queue<Transition> _currentTransactionTransitions;
227 private Transaction _currentTransaction;
228 private ulong _lastCommittedTransactionId;
229
230 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
231     : base(links)
232 {
233     if (string.IsNullOrEmpty(logAddress))
234     {
235         throw new ArgumentNullException(nameof(logAddress));
236     }
237     // В первой строке файла хранится последняя закоммиченную транзакцию.
238     // При запуске это используется для проверки удачного закрытия файла лога.
239     // In the first line of the file the last committed transaction is stored.
240     // On startup, this is used to check that the log file is successfully closed.
241     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
242     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
243     if (!lastCommittedTransition.Equals(lastWrittenTransition))
244     {
245         Dispose();
246         throw new NotSupportedException("Database is damaged, autorecovery is not
247             ↳ supported yet.");
248     }
249     if (lastCommittedTransition.Equals(default(Transition)))
250     {
251         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
252     }
253     _lastCommittedTransition = lastCommittedTransition;
254     // TODO: Think about a better way to calculate or store this value
255     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
256     _lastCommittedTransactionId = allTransitions.Max(x => x.TransactionId);
257     _uniqueTimestampFactory = new UniqueTimestampFactory();
258     _logAddress = logAddress;
259     _log = FileHelpers.Append(logAddress);
260     _transitions = new Queue<Transition>();
261     _transitionsPusher = new Task(TransitionsPusher);
262     _transitionsPusher.Start();
263 }
264
265 public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
266
267 public override ulong Create()
268 {
269     var createdLinkIndex = Links.Create();
270     var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
271     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
272         ↳ default, createdLink));
273     return createdLinkIndex;
274 }
275
276 public override ulong Update(IList<ulong> parts)
277 {

```

```

276     var linkIndex = parts[Constants.IndexPart];
277     var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
278     linkIndex = Links.Update(parts);
279     var afterLink = new UInt64Link(Links.GetLink(linkIndex));
280     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪ beforeLink, afterLink));
281     return linkIndex;
282 }
283
284 public override void Delete(ulong link)
285 {
286     var deletedLink = new UInt64Link(Links.GetLink(link));
287     Links.Delete(link);
288     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪ deletedLink, default));
289 }
290
291 [MethodImpl(MethodImplOptions.AggressiveInlining)]
292 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
    ↪ _transitions;
293
294 private void CommitTransition(Transition transition)
295 {
296     if (_currentTransaction != null)
297     {
298         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
299     }
300     var transitions = GetCurrentTransitions();
301     transitions.Enqueue(transition);
302 }
303
304 private void RevertTransition(Transition transition)
305 {
306     if (transition.After.IsNull()) // Revert Deletion with Creation
307     {
308         Links.Create();
309     }
310     else if (transition.Before.IsNull()) // Revert Creation with Deletion
311     {
312         Links.Delete(transition.After.Index);
313     }
314     else // Revert Update
315     {
316         Links.Update(new[] { transition.After.Index, transition.Before.Source,
    ↪ transition.Before.Target });
317     }
318 }
319
320 private void ResetCurrentTransation()
321 {
322     _currentTransactionId = 0;
323     _currentTransactionTransitions = null;
324     _currentTransaction = null;
325 }
326
327 private void PushTransitions()
328 {
329     if (_log == null || _transitions == null)
330     {
331         return;
332     }
333     for (var i = 0; i < _transitions.Count; i++)
334     {
335         var transition = _transitions.Dequeue();
336
337         _log.Write(transition);
338         _lastCommittedTransition = transition;
339     }
340 }
341
342 private void TransitionsPusher()
343 {
344     while (!IsDisposed && _transitionsPusher != null)
345     {
346         Thread.Sleep(DefaultPushDelay);
347         PushTransitions();
348     }
349 }
350

```

```

351     public Transaction BeginTransaction() => new Transaction(this);
352
353     private void DisposeTransitions()
354     {
355         try
356         {
357             var pusher = _transitionsPusher;
358             if (pusher != null)
359             {
360                 _transitionsPusher = null;
361                 pusher.Wait();
362             }
363             if (_transitions != null)
364             {
365                 PushTransitions();
366             }
367             _log.DisposeIfPossible();
368             FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
369         }
370         catch
371         {
372         }
373     }
374
375     #region DisposalBase
376
377     protected override void Dispose(bool manual, bool wasDisposed)
378     {
379         if (!wasDisposed)
380         {
381             DisposeTransitions();
382         }
383         base.Dispose(manual, wasDisposed);
384     }
385
386     #endregion
387 }
388 }

```

#### ./Platform.Data.Doublets.Tests/ComparisonTests.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Xunit;
4  using Platform.Diagnostics;
5
6  namespace Platform.Data.Doublets.Tests
7  {
8      public static class ComparisonTests
9      {
10         protected class UInt64Comparer : IComparer

```

```

41         result = comparer1.Compare(x, y) >= 0;
42     }
43 });
44
45 Func<ulong, ulong, int> compareReference = comparer1.Compare;
46
47 var ts3 = Performance.Measure(() =>
48 {
49     for (int i = 0; i < N; i++)
50     {
51         result = compareReference(x, y) >= 0;
52     }
53 });
54
55 var comparer2 = new UInt64Comparer();
56
57 var ts4 = Performance.Measure(() =>
58 {
59     for (int i = 0; i < N; i++)
60     {
61         result = comparer2.Compare(x, y) >= 0;
62     }
63 });
64
65 Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");
66 }
67 }
68 }

```

./Platform.Data.Doublets.Tests/DoubletLinksTests.cs

```

1  using System.Collections.Generic;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Numbers;
5  using Platform.Memory;
6  using Platform.Scopes;
7  using Platform.Setters;
8  using Platform.Data.Doublets.ResizableDirectMemory;
9
10 namespace Platform.Data.Doublets.Tests
11 {
12     public static class DoubletLinksTests
13     {
14         [Fact]
15         public static void UInt64CRUDTest()
16         {
17             using (var scope = new Scope<Types<HeapResizableDirectMemory,
18                 ↳ ResizableDirectMemoryLinks<ulong>>>())
19             {
20                 scope.Use<ILinks<ulong>>().TestCRUDOperations();
21             }
22
23             [Fact]
24             public static void UInt32CRUDTest()
25             {
26                 using (var scope = new Scope<Types<HeapResizableDirectMemory,
27                 ↳ ResizableDirectMemoryLinks<uint>>>())
28                 {
29                     scope.Use<ILinks<uint>>().TestCRUDOperations();
30                 }
31
32                 [Fact]
33                 public static void UInt16CRUDTest()
34                 {
35                     using (var scope = new Scope<Types<HeapResizableDirectMemory,
36                     ↳ ResizableDirectMemoryLinks<ushort>>>())
37                     {
38                         scope.Use<ILinks<ushort>>().TestCRUDOperations();
39                     }
40
41                     [Fact]
42                     public static void UInt8CRUDTest()
43                     {
44                         using (var scope = new Scope<Types<HeapResizableDirectMemory,
45                         ↳ ResizableDirectMemoryLinks<byte>>>())

```

```

46         scope.Use<ILinks<byte>>>().TestCRUDOperations();
47     }
48 }
49
50 private static void TestCRUDOperations<T>(this ILinks<T> links)
51 {
52     var constants = links.Constants;
53
54     var equalityComparer = EqualityComparer<T>.Default;
55
56     // Create Link
57     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
58
59     var setter = new Setter<T>(constants.Null);
60     links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
61
62     Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
63
64     var linkAddress = links.Create();
65
66     var link = new Link<T>(links.GetLink(linkAddress));
67
68     Assert.True(link.Count == 3);
69     Assert.True(equalityComparer.Equals(link.Index, linkAddress));
70     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
71     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
72
73     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.One));
74
75     // Get first link
76     setter = new Setter<T>(constants.Null);
77     links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
78
79     Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
80
81     // Update link to reference itself
82     links.Update(linkAddress, linkAddress, linkAddress);
83
84     link = new Link<T>(links.GetLink(linkAddress));
85
86     Assert.True(equalityComparer.Equals(link.Source, linkAddress));
87     Assert.True(equalityComparer.Equals(link.Target, linkAddress));
88
89     // Update link to reference null (prepare for delete)
90     var updated = links.Update(linkAddress, constants.Null, constants.Null);
91
92     Assert.True(equalityComparer.Equals(updated, linkAddress));
93
94     link = new Link<T>(links.GetLink(linkAddress));
95
96     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
97     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
98
99     // Delete link
100    links.Delete(linkAddress);
101
102    Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
103
104    setter = new Setter<T>(constants.Null);
105    links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
106
107    Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
108 }
109
110 [Fact]
111 public static void UInt64RawNumbersCRUDTest()
112 {
113     using (var scope = new Scope<Types<HeapResizableDirectMemory,
114         ↳ ResizableDirectMemoryLinks<ulong>>>()
115     {
116         {
117             scope.Use<ILinks<ulong>>>().TestRawNumbersCRUDOperations();
118         }
119     })
120 }
121
122 [Fact]
123 public static void UInt32RawNumbersCRUDTest()
124 {
125     using (var scope = new Scope<Types<HeapResizableDirectMemory,
126         ↳ ResizableDirectMemoryLinks<uint>>>()
127     {

```

```

124         scope.Use<ILinks<uint>>>().TestRawNumbersCRUDOperations();
125     }
126 }
127
128 [Fact]
129 public static void UInt16RawNumbersCRUDTest()
130 {
131     using (var scope = new Scope<Types<HeapResizableDirectMemory,
132         ↳ ResizableDirectMemoryLinks<ushort>>>())
133     {
134         scope.Use<ILinks<ushort>>>().TestRawNumbersCRUDOperations();
135     }
136 }
137
138 [Fact]
139 public static void UInt8RawNumbersCRUDTest()
140 {
141     using (var scope = new Scope<Types<HeapResizableDirectMemory,
142         ↳ ResizableDirectMemoryLinks<byte>>>())
143     {
144         scope.Use<ILinks<byte>>>().TestRawNumbersCRUDOperations();
145     }
146 }
147
148 private static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
149 {
150     // Constants
151     var constants = links.Constants;
152     var equalityComparer = EqualityComparer<T>.Default;
153
154     var h106E = new Hybrid<T>(106L, isExternal: true);
155     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
156     var h108E = new Hybrid<T>(-108L);
157
158     Assert.Equal(106L, h106E.AbsoluteValue);
159     Assert.Equal(107L, h107E.AbsoluteValue);
160     Assert.Equal(108L, h108E.AbsoluteValue);
161
162     // Create Link (External -> External)
163     var linkAddress1 = links.Create();
164
165     links.Update(linkAddress1, h106E, h108E);
166
167     var link1 = new Link<T>(links.GetLink(linkAddress1));
168
169     Assert.True(equalityComparer.Equals(link1.Source, h106E));
170     Assert.True(equalityComparer.Equals(link1.Target, h108E));
171
172     // Create Link (Internal -> External)
173     var linkAddress2 = links.Create();
174
175     links.Update(linkAddress2, linkAddress1, h108E);
176
177     var link2 = new Link<T>(links.GetLink(linkAddress2));
178
179     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
180     Assert.True(equalityComparer.Equals(link2.Target, h108E));
181
182     // Create Link (Internal -> Internal)
183     var linkAddress3 = links.Create();
184
185     links.Update(linkAddress3, linkAddress1, linkAddress2);
186
187     var link3 = new Link<T>(links.GetLink(linkAddress3));
188
189     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
190     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
191
192     // Search for created link
193     var setter1 = new Setter<T>(constants.Null);
194     links.Each(h106E, h108E, setter1.SetAndReturnFalse);
195
196     Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
197
198     // Search for nonexistent link
199     var setter2 = new Setter<T>(constants.Null);
200     links.Each(h106E, h107E, setter2.SetAndReturnFalse);
201
202     Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));

```

```

201
202 // Update link to reference null (prepare for delete)
203 var updated = links.Update(linkAddress3, constants.Null, constants.Null);
204
205 Assert.True(equalityComparer.Equals(updated, linkAddress3));
206
207 link3 = new Link<T>(links.GetLink(linkAddress3));
208
209 Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
210 Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
211
212 // Delete link
213 links.Delete(linkAddress3);
214
215 Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));
216
217 var setter3 = new Setter<T>(constants.Null);
218 links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
219
220 Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
221 }
222
223 // TODO: Test layers
224 }
225 }

```

# ./Platform.Data.Doublets.Tests/EqualityTests.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Xunit;
4 using Platform.Diagnostics;
5
6 namespace Platform.Data.Doublets.Tests
7 {
8     public static class EqualityTests
9     {
10         protected class UInt64EqualityComparer : IEqualityComparer<ulong>
11         {
12             public bool Equals(ulong x, ulong y) => x == y;
13
14             public int GetHashCode(ulong obj) => obj.GetHashCode();
15         }
16
17         private static bool Equals1<T>(T x, T y) => Equals(x, y);
18
19         private static bool Equals2<T>(T x, T y) => x.Equals(y);
20
21         private static bool Equals3(ulong x, ulong y) => x == y;
22
23         [Fact]
24         public static void EqualsPerformanceTest()
25         {
26             const int N = 1000000;
27
28             ulong x = 10;
29             ulong y = 500;
30
31             bool result = false;
32
33             var ts1 = Performance.Measure(() =>
34             {
35                 for (int i = 0; i < N; i++)
36                 {
37                     result = Equals1(x, y);
38                 }
39             });
40
41             var ts2 = Performance.Measure(() =>
42             {
43                 for (int i = 0; i < N; i++)
44                 {
45                     result = Equals2(x, y);
46                 }
47             });
48
49             var ts3 = Performance.Measure(() =>
50             {
51                 for (int i = 0; i < N; i++)
52                 {
53                     result = Equals3(x, y);

```



```

54     }
55 });
56
57 var equalityComparer1 = EqualityComparer<ulong>.Default;
58
59 var ts4 = Performance.Measure(() =>
60 {
61     for (int i = 0; i < N; i++)
62     {
63         result = equalityComparer1.Equals(x, y);
64     }
65 });
66
67 var equalityComparer2 = new UInt64EqualityComparer();
68
69 var ts5 = Performance.Measure(() =>
70 {
71     for (int i = 0; i < N; i++)
72     {
73         result = equalityComparer2.Equals(x, y);
74     }
75 });
76
77 Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;
78
79 var ts6 = Performance.Measure(() =>
80 {
81     for (int i = 0; i < N; i++)
82     {
83         result = equalityComparer3(x, y);
84     }
85 });
86
87 var comparer = Comparer<ulong>.Default;
88
89 var ts7 = Performance.Measure(() =>
90 {
91     for (int i = 0; i < N; i++)
92     {
93         result = comparer.Compare(x, y) == 0;
94     }
95 });
96
97 Assert.True(ts2 < ts1);
98 Assert.True(ts3 < ts2);
99 Assert.True(ts5 < ts4);
100 Assert.True(ts5 < ts6);
101
102 Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");
103 }
104 }
105 }

```

./Platform.Data.Doublets.Tests/LinksTests.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.IO;
5  using System.Text;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Xunit;
9  using Platform.Disposables;
10 using Platform.IO;
11 using Platform.Ranges;
12 using Platform.Random;
13 using Platform.Timestamps;
14 using Platform.Singletons;
15 using Platform.Counters;
16 using Platform.Diagnostics;
17 using Platform.Data.Constants;
18 using Platform.Data.Doublets.ResizableDirectMemory;
19 using Platform.Data.Doublets.Decorators;
20
21 namespace Platform.Data.Doublets.Tests
22 {
23     public static class LinksTests
24     {
25         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
26             ⇨ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
27     }
28 }

```

```

27 private const long Iterations = 10 * 1024;
28
29 #region Concept
30
31 [Fact]
32 public static void MultipleCreateAndDeleteTest()
33 {
34     //const int N = 21;
35
36     using (var scope = new TempLinksTestScope())
37     {
38         var links = scope.Links;
39
40         for (var N = 0; N < 100; N++)
41         {
42             var random = new System.Random(N);
43
44             var created = 0;
45             var deleted = 0;
46
47             for (var i = 0; i < N; i++)
48             {
49                 var linksCount = links.Count();
50
51                 var createPoint = random.NextBoolean();
52
53                 if (linksCount > 2 && createPoint)
54                 {
55                     var linksAddressRange = new Range<ulong>(1, linksCount);
56                     var source = random.NextUInt64(linksAddressRange);
57                     var target = random.NextUInt64(linksAddressRange); //-V3086
58
59                     var resultLink = links.CreateAndUpdate(source, target);
60                     if (resultLink > linksCount)
61                     {
62                         created++;
63                     }
64                 }
65                 else
66                 {
67                     links.Create();
68                     created++;
69                 }
70             }
71
72             Assert.True(created == (int)links.Count());
73
74             for (var i = 0; i < N; i++)
75             {
76                 var link = (ulong)i + 1;
77                 if (links.Exists(link))
78                 {
79                     links.Delete(link);
80                     deleted++;
81                 }
82             }
83
84             Assert.True(links.Count() == 0);
85         }
86     }
87 }
88
89 [Fact]
90 public static void CascadeUpdateTest()
91 {
92     var itself = _constants.Itself;
93
94     using (var scope = new TempLinksTestScope(useLog: true))
95     {
96         var links = scope.Links;
97
98         var l1 = links.Create();
99         var l2 = links.Create();
100
101         l2 = links.Update(l2, l2, l1, l2);
102
103         links.CreateAndUpdate(l2, itself);
104         links.CreateAndUpdate(l2, itself);
105
106         l2 = links.Update(l2, l1);

```

```

107         links.Delete(l2);
108
109         Global.Trash = links.Count();
110
111         links.Unsync.DisposeIfPossible(); // Close links to access log
112
113         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope ↵
114             ↪ e.TempTransactionLogFilename);
115     }
116 }
117
118 [Fact]
119 public static void BasicTransactionLogTest()
120 {
121     using (var scope = new TempLinksTestScope(useLog: true))
122     {
123         var links = scope.Links;
124         var l1 = links.Create();
125         var l2 = links.Create();
126
127         Global.Trash = links.Update(l2, l2, l1, l2);
128
129         links.Delete(l1);
130
131         links.Unsync.DisposeIfPossible(); // Close links to access log
132
133         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope ↵
134             ↪ e.TempTransactionLogFilename);
135     }
136 }
137
138 [Fact]
139 public static void TransactionAutoRevertedTest()
140 {
141     // Auto Reverted (Because no commit at transaction)
142     using (var scope = new TempLinksTestScope(useLog: true))
143     {
144         var links = scope.Links;
145         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
146         using (var transaction = transactionsLayer.BeginTransaction())
147         {
148             var l1 = links.Create();
149             var l2 = links.Create();
150
151             links.Update(l2, l2, l1, l2);
152         }
153
154         Assert.Equal(0UL, links.Count());
155
156         links.Unsync.DisposeIfPossible();
157
158         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s ↵
159             ↪ cope.TempTransactionLogFilename);
160         Assert.Single(transitions);
161     }
162 }
163
164 [Fact]
165 public static void TransactionUserCodeErrorNoDataSavedTest()
166 {
167     // User Code Error (Autoreverted), no data saved
168     var itself = _constants.Itself;
169
170     TempLinksTestScope lastScope = null;
171     try
172     {
173         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false, ↵
174             ↪ useLog: true))
175         {
176             var links = scope.Links;
177             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor ↵
178                 ↪ atorBase<ulong>)links.Unsync).Links;
179             using (var transaction = transactionsLayer.BeginTransaction())
180             {
181                 var l1 = links.CreateAndUpdate(itself, itself);
182                 var l2 = links.CreateAndUpdate(itself, itself);
183
184                 l2 = links.Update(l2, l2, l1, l2);
185             }
186         }
187     }
188     catch { }
189 }

```

```

181         links.CreateAndUpdate(l2, itself);
182         links.CreateAndUpdate(l2, itself);
183
184         //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi_
185         ↪ tion>(scope.TempTransactionLogFilename);
186
187         l2 = links.Update(l2, l1);
188
189         links.Delete(l2);
190
191         ExceptionThrower();
192
193         transaction.Commit();
194     }
195
196     Global.Trash = links.Count();
197 }
198
199 catch
200 {
201     Assert.False(lastScope == null);
202
203     var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l_
204     ↪ astScope.TempTransactionLogFilename);
205
206     Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
207     ↪ transitions[0].After.IsNull());
208
209     lastScope.DeleteFiles();
210 }
211
212 [Fact]
213 public static void TransactionUserCodeErrorSomeDataSavedTest()
214 {
215     // User Code Error (Autoreverted), some data saved
216     var itself = _constants.Itself;
217
218     TempLinksTestScope lastScope = null;
219     try
220     {
221         ulong l1;
222         ulong l2;
223
224         using (var scope = new TempLinksTestScope(useLog: true))
225         {
226             var links = scope.Links;
227             l1 = links.CreateAndUpdate(itself, itself);
228             l2 = links.CreateAndUpdate(itself, itself);
229
230             l2 = links.Update(l2, l2, l1, l2);
231
232             links.CreateAndUpdate(l2, itself);
233             links.CreateAndUpdate(l2, itself);
234
235             links.Unsync.DisposeIfPossible();
236
237             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(_
238             ↪ scope.TempTransactionLogFilename);
239         }
240
241         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
242         ↪ useLog: true))
243         {
244             var links = scope.Links;
245             var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
246             using (var transaction = transactionsLayer.BeginTransaction())
247             {
248                 l2 = links.Update(l2, l1);
249
250                 links.Delete(l2);
251
252                 ExceptionThrower();
253
254                 transaction.Commit();
255             }
256
257             Global.Trash = links.Count();
258         }
259     }
260 }

```

```

256     }
257     catch
258     {
259         Assert.False(lastScope == null);
260
261         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
            ↪ Scope.TempTransactionLogFilename);
262
263         lastScope.DeleteFiles();
264     }
265 }
266
267 [Fact]
268 public static void TransactionCommit()
269 {
270     var itself = _constants.Itself;
271
272     var tempDatabaseFilename = Path.GetTempFileName();
273     var tempTransactionLogFilename = Path.GetTempFileName();
274
275     // Commit
276     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
            ↪ tempTransactionLogFilename))
277     using (var links = new UInt64Links(memoryAdapter))
278     {
279         using (var transaction = memoryAdapter.BeginTransaction())
280         {
281             var l1 = links.CreateAndUpdate(itself, itself);
282             var l2 = links.CreateAndUpdate(itself, itself);
283
284             Global.Trash = links.Update(l2, l2, l1, l2);
285
286             links.Delete(l1);
287
288             transaction.Commit();
289         }
290
291         Global.Trash = links.Count();
292     }
293
294     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
            ↪ sactionLogFilename);
295 }
296
297 [Fact]
298 public static void TransactionDamage()
299 {
300     var itself = _constants.Itself;
301
302     var tempDatabaseFilename = Path.GetTempFileName();
303     var tempTransactionLogFilename = Path.GetTempFileName();
304
305     // Commit
306     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
            ↪ tempTransactionLogFilename))
307     using (var links = new UInt64Links(memoryAdapter))
308     {
309         using (var transaction = memoryAdapter.BeginTransaction())
310         {
311             var l1 = links.CreateAndUpdate(itself, itself);
312             var l2 = links.CreateAndUpdate(itself, itself);
313
314             Global.Trash = links.Update(l2, l2, l1, l2);
315
316             links.Delete(l1);
317
318             transaction.Commit();
319         }
320
321         Global.Trash = links.Count();
322     }
323
324     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
            ↪ sactionLogFilename);
325
326     // Damage database
327

```

```

328 FileHelpers.WriteFirst(tempTransactionLogFilename, new
    ↳ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
329
330 // Try load damaged database
331 try
332 {
333     // TODO: Fix
334     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
        ↳ tempTransactionLogFilename))
335     using (var links = new UInt64Links(memoryAdapter))
336     {
337         Global.Trash = links.Count();
338     }
339 }
340 catch (NotSupportedException ex)
341 {
342     Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
        ↳ yet.");
343 }
344
345 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
    ↳ sactionLogFilename);
346
347 File.Delete(tempDatabaseFilename);
348 File.Delete(tempTransactionLogFilename);
349 }
350
351 [Fact]
352 public static void Bug1Test()
353 {
354     var tempDatabaseFilename = Path.GetTempFileName();
355     var tempTransactionLogFilename = Path.GetTempFileName();
356
357     var itself = _constants.Itself;
358
359     // User Code Error (Autoreverted), some data saved
360     try
361     {
362         ulong l1;
363         ulong l2;
364
365         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
            ↳ tempTransactionLogFilename))
366         using (var links = new UInt64Links(memoryAdapter))
367         {
368             l1 = links.CreateAndUpdate(itself, itself);
369             l2 = links.CreateAndUpdate(itself, itself);
370
371             l2 = links.Update(l2, l2, l1, l2);
372
373             links.CreateAndUpdate(l2, itself);
374             links.CreateAndUpdate(l2, itself);
375         }
376
377         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
            ↳ TransactionLogFilename);
378
379         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
            ↳ tempTransactionLogFilename))
380         using (var links = new UInt64Links(memoryAdapter))
381         {
382             using (var transaction = memoryAdapter.BeginTransaction())
383             {
384                 l2 = links.Update(l2, l1);
385
386                 links.Delete(l2);
387
388                 ExceptionThrower();
389
390                 transaction.Commit();
391             }
392
393             Global.Trash = links.Count();
394         }
395     }
396     catch

```

```

397     {
398         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp ↵
        ↵ TransactionLogFilename);
399     }
400
401     File.Delete(tempDatabaseFilename);
402     File.Delete(tempTransactionLogFilename);
403 }
404
405 private static void ExceptionThrower()
406 {
407     throw new Exception();
408 }
409
410 [Fact]
411 public static void PathsTest()
412 {
413     var source = _constants.SourcePart;
414     var target = _constants.TargetPart;
415
416     using (var scope = new TempLinksTestScope())
417     {
418         var links = scope.Links;
419         var l1 = links.CreatePoint();
420         var l2 = links.CreatePoint();
421
422         var r1 = links.GetByKeys(l1, source, target, source);
423         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
424     }
425 }
426
427 [Fact]
428 public static void RecursiveStringFormattingTest()
429 {
430     using (var scope = new TempLinksTestScope(useSequences: true))
431     {
432         var links = scope.Links;
433         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
434
435         var a = links.CreatePoint();
436         var b = links.CreatePoint();
437         var c = links.CreatePoint();
438
439         var ab = links.CreateAndUpdate(a, b);
440         var cb = links.CreateAndUpdate(c, b);
441         var ac = links.CreateAndUpdate(a, c);
442
443         a = links.Update(a, c, b);
444         b = links.Update(b, a, c);
445         c = links.Update(c, a, b);
446
447         Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
448         Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
449         Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
450
451         Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
        ↵ "(5:(4:5 (6:5 4)) 6)");
452         Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
        ↵ "(6:(5:(4:5 6) 6) 4)");
453         Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
        ↵ "(4:(5:4 (6:5 4)) 6)");
454
455         // TODO: Think how to build balanced syntax tree while formatting structure (eg.
        ↵ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
456
457         Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
        ↵ "{5}{5}{4}{6}");
458         Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
        ↵ "{5}{6}{6}{4}");
459         Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
        ↵ "{4}{5}{4}{6}");
460     }
461 }
462
463 private static void DefaultFormatter(StringBuilder sb, ulong link)
464 {
465     sb.Append(link.ToString());
466 }

```

```
467 #endregion
```

```
468 #region Performance
```

```
471 /*
472 public static void RunAllPerformanceTests()
473 {
474     try
475     {
476         links.TestLinksInSteps();
477     }
478     catch (Exception ex)
479     {
480         ex.WriteToConsole();
481     }
482     return;
483
484     try
485     {
486         //ThreadPool.SetMaxThreads(2, 2);
487
488         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
489         // результат
490         // Также это дополнительно помогает в отладке
491         // Увеличивает вероятность попадания информации в кэши
492         for (var i = 0; i < 10; i++)
493         {
494             //0 - 10 ГБ
495             //Каждые 100 МБ срез цифр
496
497             //links.TestGetSourceFunction();
498             //links.TestGetSourceFunctionInParallel();
499             //links.TestGetTargetFunction();
500             //links.TestGetTargetFunctionInParallel();
501             links.Create64BillionLinks();
502
503             links.TestRandomSearchFixed();
504             //links.Create64BillionLinksInParallel();
505             links.TestEachFunction();
506             //links.TestForeach();
507             //links.TestParallelForeach();
508         }
509
510         links.TestDeletionOfAllLinks();
511
512     }
513     catch (Exception ex)
514     {
515         ex.WriteToConsole();
516     }
517 }*/
518
519 /*
520 public static void TestLinksInSteps()
521 {
522     const long gibibyte = 1024 * 1024 * 1024;
523     const long mebibyte = 1024 * 1024;
524
525     var totalLinksToCreate = gibibyte /
526     Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
527     var linksStep = 102 * mebibyte /
528     Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
529
530     var creationMeasurements = new List<TimeSpan>();
531     var searchMeasurements = new List<TimeSpan>();
532     var deletionMeasurements = new List<TimeSpan>();
533
534     GetBaseRandomLoopOverhead(linksStep);
535     GetBaseRandomLoopOverhead(linksStep);
536
537     var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
538
539     ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
540
541     var loops = totalLinksToCreate / linksStep;
542
543     for (int i = 0; i < loops; i++)
544     {
```



```

544         creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
545         searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
546
547         Console.WriteLine("\rC + S {0}/{1}", i + 1, loops);
548     }
549
550     ConsoleHelpers.Debug();
551
552     for (int i = 0; i < loops; i++)
553     {
554         deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
555
556         Console.WriteLine("\rD {0}/{1}", i + 1, loops);
557     }
558
559     ConsoleHelpers.Debug();
560
561     ConsoleHelpers.Debug("C S D");
562
563     for (int i = 0; i < loops; i++)
564     {
565         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↪ searchMeasurements[i], deletionMeasurements[i]);
566     }
567
568     ConsoleHelpers.Debug("C S D (no overhead)");
569
570     for (int i = 0; i < loops; i++)
571     {
572         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
↪ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
573     }
574
575     ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↪ links.Total);
576 }
577
578 private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
↪ amountToCreate)
579 {
580     for (long i = 0; i < amountToCreate; i++)
581         links.Create(0, 0);
582 }
583
584 private static TimeSpan GetBaseRandomLoopOverhead(long loops)
585 {
586     return Measure(() =>
587     {
588         ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
589         ulong result = 0;
590         for (long i = 0; i < loops; i++)
591         {
592             var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
593             var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
594
595             result += maxValue + source + target;
596         }
597         Global.Trash = result;
598     });
599 }
600 */
601
602 [Fact(Skip = "performance test")]
603 public static void GetSourceTest()
604 {
605     using (var scope = new TempLinksTestScope())
606     {
607         var links = scope.Links;
608         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
↪ Iterations);
609
610         ulong counter = 0;
611
612         //var firstLink = links.First();
613         // Создаём одну связь, из которой будет производить считывание
614         var firstLink = links.Create();
615
616         var sw = Stopwatch.StartNew();
617

```

```

618 // Тестируем саму функцию
619 for (ulong i = 0; i < Iterations; i++)
620 {
621     counter += links.GetSource(firstLink);
622 }
623
624 var elapsedTime = sw.Elapsed;
625
626 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
627
628 // Удаляем связь, из которой производилось считывание
629 links.Delete(firstLink);
630
631 ConsoleHelpers.Debug(
632     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
633 }
634
635 }
636
637 [Fact(Skip = "performance test")]
638 public static void GetSourceInParallel()
639 {
640     using (var scope = new TempLinksTestScope())
641     {
642         var links = scope.Links;
643         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
        ↳ parallel.", Iterations);
644
645         long counter = 0;
646
647         //var firstLink = links.First();
648         var firstLink = links.Create();
649
650         var sw = Stopwatch.StartNew();
651
652         // Тестируем саму функцию
653         Parallel.For(0, Iterations, x =>
654         {
655             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
656             //Interlocked.Increment(ref counter);
657         });
658
659         var elapsedTime = sw.Elapsed;
660
661         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
662
663         links.Delete(firstLink);
664
665         ConsoleHelpers.Debug(
666             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
667     }
668 }
669
670
671 [Fact(Skip = "performance test")]
672 public static void TestGetTarget()
673 {
674     using (var scope = new TempLinksTestScope())
675     {
676         var links = scope.Links;
677         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
        ↳ Iterations);
678
679         ulong counter = 0;
680
681         //var firstLink = links.First();
682         var firstLink = links.Create();
683
684         var sw = Stopwatch.StartNew();
685
686         for (ulong i = 0; i < Iterations; i++)
687         {
688             counter += links.GetTarget(firstLink);
689         }
690
691         var elapsedTime = sw.Elapsed;
692
693         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

```

```

694         links.Delete(firstLink);
695
696     ConsoleHelpers.Debug(
697         "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
698         ↳ second), counter result: {3}",
699         Iterations, elapsedTime, (long)iterationsPerSecond, counter);
700     }
701 }
702
703 [Fact(Skip = "performance test")]
704 public static void TestGetTargetInParallel()
705 {
706     using (var scope = new TempLinksTestScope())
707     {
708         var links = scope.Links;
709         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
710         ↳ parallel.", Iterations);
711
712         long counter = 0;
713
714         //var firstLink = links.First();
715         var firstLink = links.Create();
716
717         var sw = Stopwatch.StartNew();
718
719         Parallel.For(0, Iterations, x =>
720         {
721             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
722             //Interlocked.Increment(ref counter);
723         });
724
725         var elapsedTime = sw.Elapsed;
726
727         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
728
729         links.Delete(firstLink);
730
731         ConsoleHelpers.Debug(
732             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
733             ↳ second), counter result: {3}",
734             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
735     }
736 }
737
738 // TODO: Заполнить базу данных перед тестом
739 /*
740 [Fact]
741 public void TestRandomSearchFixed()
742 {
743     var tempFilename = Path.GetTempFileName();
744
745     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
746     ↳ DefaultLinksSizeStep))
747     {
748         long iterations = 64 * 1024 * 1024 /
749         ↳ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
750
751         ulong counter = 0;
752         var maxLink = links.Total;
753
754         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
755
756         var sw = Stopwatch.StartNew();
757
758         for (var i = iterations; i > 0; i--)
759         {
760             var source =
761             ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
762             var target =
763             ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
764
765             counter += links.Search(source, target);
766         }
767
768         var elapsedTime = sw.Elapsed;
769
770         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
771     }
772 }

```

```

766         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
↪ counter);
767     }
768
769     File.Delete(tempFilename);
770 }*/
771
772 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
773 public static void TestRandomSearchAll()
774 {
775     using (var scope = new TempLinksTestScope())
776     {
777         var links = scope.Links;
778         ulong counter = 0;
779
780         var maxLink = links.Count();
781
782         var iterations = links.Count();
783
784         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
↪ links.Count());
785
786         var sw = Stopwatch.StartNew();
787
788         for (var i = iterations; i > 0; i--)
789         {
790             var linksAddressRange = new Range<ulong>(_constants.MinPossibleIndex,
↪ maxLink);
791
792             var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
793             var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
794
795             counter += links.SearchOrDefault(source, target);
796         }
797
798         var elapsedTime = sw.Elapsed;
799
800         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
801
802         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}",
↪ iterations, elapsedTime, (long)iterationsPerSecond, counter);
803     }
804 }
805
806 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
807 public static void TestEach()
808 {
809     using (var scope = new TempLinksTestScope())
810     {
811         var links = scope.Links;
812
813         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
814
815         ConsoleHelpers.Debug("Testing Each function.");
816
817         var sw = Stopwatch.StartNew();
818
819         links.Each(counter.IncrementAndReturnTrue);
820
821         var elapsedTime = sw.Elapsed;
822
823         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
824
825         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
↪ links per second)",
↪ counter, elapsedTime, (long)linksPerSecond);
826     }
827 }
828
829 }
830
831 /*
832 [Fact]
833 public static void TestForeach()
834 {
835     var tempFilename = Path.GetTempFileName();
836
837     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪ DefaultLinksSizeStep))

```

```

838     {
839         ulong counter = 0;
840
841         ConsoleHelpers.Debug("Testing foreach through links.");
842
843         var sw = Stopwatch.StartNew();
844
845         //foreach (var link in links)
846         //{
847             counter++;
848         //}
849
850         var elapsedTime = sw.Elapsed;
851
852         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
853
854         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪ links per second)", counter, elapsedTime, (long)linksPerSecond);
855     }
856
857     File.Delete(tempFilename);
858 }
859 */
860
861 /*
862 [Fact]
863 public static void TestParallelForeach()
864 {
865     var tempFilename = Path.GetTempFileName();
866
867     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪ DefaultLinksSizeStep))
868     {
869         long counter = 0;
870
871         ConsoleHelpers.Debug("Testing parallel foreach through links.");
872
873         var sw = Stopwatch.StartNew();
874
875         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
876         //{
877             Interlocked.Increment(ref counter);
878         //});
879
880         var elapsedTime = sw.Elapsed;
881
882         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
883
884         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
885     }
886
887     File.Delete(tempFilename);
888 }
889 */
890
891 [Fact(Skip = "performance test")]
892 public static void Create64BillionLinks()
893 {
894     using (var scope = new TempLinksTestScope())
895     {
896         var links = scope.Links;
897         var linksBeforeTest = links.Count();
898
899         long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
900
901         ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
902
903         var elapsedTime = Performance.Measure(() =>
904         {
905             for (long i = 0; i < linksToCreate; i++)
906             {
907                 links.Create();
908             }
909         });
910
911         var linksCreated = links.Count() - linksBeforeTest;
912         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
913

```

```

914         ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
915
916         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
917             ↪ linksCreated, elapsedTime,
918             (long)linksPerSecond);
919     }
920 }
921
922 [Fact(Skip = "performance test")]
923 public static void Create64BillionLinksInParallel()
924 {
925     using (var scope = new TempLinksTestScope())
926     {
927         var links = scope.Links;
928         var linksBeforeTest = links.Count();
929
930         var sw = Stopwatch.StartNew();
931
932         long linksToCreate = 64 * 1024 * 1024 /
933             ↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
934
935         ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
936
937         Parallel.For(0, linksToCreate, x => links.Create());
938
939         var elapsedTime = sw.Elapsed;
940
941         var linksCreated = links.Count() - linksBeforeTest;
942         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
943
944         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
945             ↪ linksCreated, elapsedTime,
946             (long)linksPerSecond);
947     }
948 }
949
950 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
951 public static void TestDeletionOfAllLinks()
952 {
953     using (var scope = new TempLinksTestScope())
954     {
955         var links = scope.Links;
956         var linksBeforeTest = links.Count();
957
958         ConsoleHelpers.Debug("Deleting all links");
959
960         var elapsedTime = Performance.Measure(links.DeleteAll);
961
962         var linksDeleted = linksBeforeTest - links.Count();
963         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
964
965         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
966             ↪ linksDeleted, elapsedTime,
967             (long)linksPerSecond);
968     }
969 }
970
971 #endregion
972 }

```

./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using Xunit;
5  using Platform.Interfaces;
6  using Platform.Data.Doublets.Sequences;
7  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
8  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
9  using Platform.Data.Doublets.Sequences.Converters;
10 using Platform.Data.Doublets.PropertyOperators;
11 using Platform.Data.Doublets.Incrementers;
12 using Platform.Data.Doublets.Converters;
13
14 namespace Platform.Data.Doublets.Tests
15 {
16     public static class OptimalVariantSequenceTests
17     {
18         private const string SequenceExample = "зеленела зелёная зелень";

```

```

19 [Fact]
20 public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
21 {
22     using (var scope = new TempLinksTestScope(useSequences: true))
23     {
24         var links = scope.Links;
25         var sequences = scope.Sequences;
26         var constants = links.Constants;
27
28         links.UseUnicode();
29
30         var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
31
32         var meaningRoot = links.CreatePoint();
33         var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
34         var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
35         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
36             ↪ constants.Itself);
37
38         var unaryNumberToAddressConveter = new
39             ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
40         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
41         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
42             ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
43         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
44             ↪ frequencyPropertyMarker, frequencyMarker);
45         var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
46             ↪ frequencyPropertyOperator, frequencyIncrementer);
47         var linkToItsFrequencyNumberConverter = new
48             ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
49             ↪ unaryNumberToAddressConveter);
50         var sequenceToItsLocalElementLevelsConverter = new
51             ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
52             ↪ linkToItsFrequencyNumberConverter);
53         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
54             ↪ sequenceToItsLocalElementLevelsConverter);
55
56         ExecuteTest(links, sequences, sequence,
57             ↪ sequenceToItsLocalElementLevelsConverter, linkFrequencyIncrementer,
58             ↪ optimalVariantConverter);
59     }
60 }
61
62 [Fact]
63 public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
64 {
65     using (var scope = new TempLinksTestScope(useSequences: true))
66     {
67         var links = scope.Links;
68         var sequences = scope.Sequences;
69
70         links.UseUnicode();
71
72         var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
73
74         var linksToFrequencies = new Dictionary<ulong, ulong>();
75
76         var totalSequenceSymbolFrequencyCounter = new
77             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
78
79         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
80             ↪ totalSequenceSymbolFrequencyCounter);
81
82         var linkFrequencyIncrementer = new
83             ↪ FrequenciesCacheBasedLinkFrequencyIncrementer<ulong>(linkFrequenciesCache);
84         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
85             ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
86
87         var sequenceToItsLocalElementLevelsConverter = new
88             ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
89             ↪ linkToItsFrequencyNumberConverter);
90         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
91             ↪ sequenceToItsLocalElementLevelsConverter);
92
93         ExecuteTest(links, sequences, sequence,
94             ↪ sequenceToItsLocalElementLevelsConverter, linkFrequencyIncrementer,
95             ↪ optimalVariantConverter);
96     }
97 }

```

```

76     }
77 }
78
79 private static void ExecuteTest(SynchronizedLinks<ulong> links, Sequences.Sequences
    ↪ sequences, ulong[] sequence, SequenceToItsLocalElementLevelsConverter<ulong>
    ↪ sequenceToItsLocalElementLevelsConverter, IIncrementer<IList<ulong>>
    ↪ linkFrequencyIncrementer, OptimalVariantConverter<ulong> optimalVariantConverter)
80 {
81     linkFrequencyIncrementer.Increment(sequence);
82
83     var levels = sequenceToItsLocalElementLevelsConverter.Convert(sequence);
84
85     var optimalVariant = optimalVariantConverter.Convert(sequence);
86
87     var readSequence1 = sequences.ReadSequenceCore(optimalVariant, links.IsPartialPoint);
88
89     Assert.True(sequence.SequenceEqual(readSequence1));
90 }
91 }
92 }

```

# ./Platform.Data.Doublets.Tests/ReadSequenceTests.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4 using System.Linq;
5 using Xunit;
6 using Platform.Data.Sequences;
7 using Platform.Data.Doublets.Sequences.Converters;
8
9 namespace Platform.Data.Doublets.Tests
10 {
11     public static class ReadSequenceTests
12     {
13         [Fact]
14         public static void ReadSequenceTest()
15         {
16             const long sequenceLength = 2000;
17
18             using (var scope = new TempLinksTestScope(useSequences: true))
19             {
20                 var links = scope.Links;
21                 var sequences = scope.Sequences;
22
23                 var sequence = new ulong[sequenceLength];
24                 for (var i = 0; i < sequenceLength; i++)
25                 {
26                     sequence[i] = links.Create();
27                 }
28
29                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
30
31                 var sw1 = Stopwatch.StartNew();
32                 var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
33
34                 var sw2 = Stopwatch.StartNew();
35                 var readSequence1 = sequences.ReadSequenceCore(balancedVariant,
    ↪ links.IsPartialPoint); sw2.Stop();
36
37                 var sw3 = Stopwatch.StartNew();
38                 var readSequence2 = new List<ulong>();
39                 SequenceWalker.WalkRight(balancedVariant,
40                                         links.GetSource,
41                                         links.GetTarget,
42                                         links.IsPartialPoint,
43                                         readSequence2.Add);
44                 sw3.Stop();
45
46                 Assert.True(sequence.SequenceEqual(readSequence1));
47
48                 Assert.True(sequence.SequenceEqual(readSequence2));
49
50                 // Assert.True(sw2.Elapsed < sw3.Elapsed);
51
52                 Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
    ↪ {sw2.Elapsed}");
53
54                 for (var i = 0; i < sequenceLength; i++)
55                 {
56                     links.Delete(sequence[i]);

```



```

57     }
58 }
59 }
60 }
61 }

```

./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs

```

1  using System.IO;
2  using Xunit;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using Platform.Data.Constants;
6  using Platform.Data.Doublets.ResizableDirectMemory;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public static class ResizableDirectMemoryLinksTests
11     {
12         private static readonly LinksCombinedConstants<ulong, ulong, int> _constants =
13             ↪ Default<LinksCombinedConstants<ulong, ulong, int>>.Instance;
14
15         [Fact]
16         public static void BasicFileMappedMemoryTest()
17         {
18             var tempFilename = Path.GetTempFileName();
19             using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
20             {
21                 memoryAdapter.TestBasicMemoryOperations();
22             }
23             File.Delete(tempFilename);
24
25         [Fact]
26         public static void BasicHeapMemoryTest()
27         {
28             using (var memory = new
29                 ↪ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
30             using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
31                 ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
32             {
33                 memoryAdapter.TestBasicMemoryOperations();
34             }
35
36         private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
37         {
38             var link = memoryAdapter.Create();
39             memoryAdapter.Delete(link);
40
41         [Fact]
42         public static void NonexistentReferencesHeapMemoryTest()
43         {
44             using (var memory = new
45                 ↪ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
46             using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
47                 ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
48             {
49                 memoryAdapter.TestNonexistentReferences();
50             }
51
52         private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
53         {
54             var link = memoryAdapter.Create();
55             memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
56             var resultLink = _constants.Null;
57             memoryAdapter.Each(foundLink =>
58             {
59                 resultLink = foundLink[_constants.IndexPart];
60                 return _constants.Break;
61             }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
62             Assert.True(resultLink == link);
63             Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
64             memoryAdapter.Delete(link);
65         }
66     }

```

## ./Platform.Data.Doublets.Tests/ScopeTests.cs

```
1  using Xunit;
2  using Platform.Scopes;
3  using Platform.Memory;
4  using Platform.Data.Doublets.ResizableDirectMemory;
5  using Platform.Data.Doublets.Decorators;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public static class ScopeTests
10     {
11         [Fact]
12         public static void SingleDependencyTest()
13         {
14             using (var scope = new Scope())
15             {
16                 scope.IncludeAssemblyOf<IMemory>();
17                 var instance = scope.Use<IDirectMemory>();
18                 Assert.IsType<HeapResizableDirectMemory>(instance);
19             }
20         }
21
22         [Fact]
23         public static void CascadeDependencyTest()
24         {
25             using (var scope = new Scope())
26             {
27                 scope.Include<TemporaryFileMappedResizableDirectMemory>();
28                 scope.Include<UInt64ResizableDirectMemoryLinks>();
29                 var instance = scope.Use<ILinks<ulong>>();
30                 Assert.IsType<UInt64ResizableDirectMemoryLinks>(instance);
31             }
32         }
33
34         [Fact]
35         public static void FullAutoResolutionTest()
36         {
37             using (var scope = new Scope(autoInclude: true, autoExplore: true))
38             {
39                 var instance = scope.Use<UInt64Links>();
40                 Assert.IsType<UInt64Links>(instance);
41             }
42         }
43     }
44 }
```

## ./Platform.Data.Doublets.Tests/SequencesTests.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Collections;
7  using Platform.Random;
8  using Platform.IO;
9  using Platform.Singletons;
10 using Platform.Data.Constants;
11 using Platform.Data.Doublets.Sequences;
12 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
13 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
14 using Platform.Data.Doublets.Sequences.Converters;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class SequencesTests
19     {
20         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
21             ↪ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
22
23         static SequencesTests()
24         {
25             // Trigger static constructor to not mess with performance measurements
26             _ = BitString.GetBitMaskFromIndex(1);
27         }
28
29         [Fact]
30         public static void CreateAllVariantsTest()
31         {
32             const long sequenceLength = 8;
```

```

33     using (var scope = new TempLinksTestScope(useSequences: true))
34     {
35         var links = scope.Links;
36         var sequences = scope.Sequences;
37
38         var sequence = new ulong[sequenceLength];
39         for (var i = 0; i < sequenceLength; i++)
40         {
41             sequence[i] = links.Create();
42         }
43
44         var sw1 = Stopwatch.StartNew();
45         var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
46
47         var sw2 = Stopwatch.StartNew();
48         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
49
50         Assert.True(results1.Count > results2.Length);
51         Assert.True(sw1.Elapsed > sw2.Elapsed);
52
53         for (var i = 0; i < sequenceLength; i++)
54         {
55             links.Delete(sequence[i]);
56         }
57
58         Assert.True(links.Count() == 0);
59     }
60 }
61
62 // [Fact]
63 // public void CUDTest()
64 // {
65 //     var tempFilename = Path.GetTempFileName();
66 //
67 //     const long sequenceLength = 8;
68 //
69 //     const ulong itself = LinksConstants.Itself;
70 //
71 //     using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
72 // ↪ DefaultLinksSizeStep))
73 //     using (var links = new Links(memoryAdapter))
74 //     {
75 //         var sequence = new ulong[sequenceLength];
76 //         for (var i = 0; i < sequenceLength; i++)
77 //             sequence[i] = links.Create(itself, itself);
78 //
79 //         SequencesOptions o = new SequencesOptions();
80 //
81 //         // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
82 //         o.
83 //
84 //
85 //         var sequences = new Sequences(links);
86 //
87 //         var sw1 = Stopwatch.StartNew();
88 //         var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
89 //
90 //         var sw2 = Stopwatch.StartNew();
91 //         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
92 //
93 //         Assert.True(results1.Count > results2.Length);
94 //         Assert.True(sw1.Elapsed > sw2.Elapsed);
95 //
96 //         for (var i = 0; i < sequenceLength; i++)
97 //             links.Delete(sequence[i]);
98 //     }
99 //
100 //     File.Delete(tempFilename);
101 // }
102
103 [Fact]
104 public static void AllVariantsSearchTest()
105 {
106     const long sequenceLength = 8;
107
108     using (var scope = new TempLinksTestScope(useSequences: true))
109     {
110         var links = scope.Links;
111         var sequences = scope.Sequences;

```

```

112     var sequence = new ulong[sequenceLength];
113     for (var i = 0; i < sequenceLength; i++)
114     {
115         sequence[i] = links.Create();
116     }
117
118     var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119
120     //for (int i = 0; i < createResults.Length; i++)
121     //    sequences.Create(createResults[i]);
122
123     var sw0 = Stopwatch.StartNew();
124     var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125
126     var sw1 = Stopwatch.StartNew();
127     var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128
129     var sw2 = Stopwatch.StartNew();
130     var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131
132     var sw3 = Stopwatch.StartNew();
133     var searchResults3 = sequences.Each(sequence); sw3.Stop();
134
135     var intersection0 = createResults.Intersect(searchResults0).ToList();
136     Assert.True(intersection0.Count == searchResults0.Count);
137     Assert.True(intersection0.Count == createResults.Length);
138
139     var intersection1 = createResults.Intersect(searchResults1).ToList();
140     Assert.True(intersection1.Count == searchResults1.Count);
141     Assert.True(intersection1.Count == createResults.Length);
142
143     var intersection2 = createResults.Intersect(searchResults2).ToList();
144     Assert.True(intersection2.Count == searchResults2.Count);
145     Assert.True(intersection2.Count == createResults.Length);
146
147     var intersection3 = createResults.Intersect(searchResults3).ToList();
148     Assert.True(intersection3.Count == searchResults3.Count);
149     Assert.True(intersection3.Count == createResults.Length);
150
151     for (var i = 0; i < sequenceLength; i++)
152     {
153         links.Delete(sequence[i]);
154     }
155 }
156
157 [Fact]
158 public static void BalancedVariantSearchTest()
159 {
160     const long sequenceLength = 200;
161
162     using (var scope = new TempLinksTestScope(useSequences: true))
163     {
164         var links = scope.Links;
165         var sequences = scope.Sequences;
166
167         var sequence = new ulong[sequenceLength];
168         for (var i = 0; i < sequenceLength; i++)
169         {
170             sequence[i] = links.Create();
171         }
172
173         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
174
175         var sw1 = Stopwatch.StartNew();
176         var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
177
178         var sw2 = Stopwatch.StartNew();
179         var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
180
181         var sw3 = Stopwatch.StartNew();
182         var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
183
184         // На количестве в 200 элементов это будет занимать вечность
185         //var sw4 = Stopwatch.StartNew();
186         //var searchResults4 = sequences.Each(sequence); sw4.Stop();
187
188         Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
189
190
191

```

```

192     Assert.True(searchResults3.Count == 1 && balancedVariant ==
193         ↳ searchResults3.First());
194
195     //Assert.True(sw1.Elapsed < sw2.Elapsed);
196
197     for (var i = 0; i < sequenceLength; i++)
198     {
199         links.Delete(sequence[i]);
200     }
201 }
202
203 [Fact]
204 public static void AllPartialVariantsSearchTest()
205 {
206     const long sequenceLength = 8;
207
208     using (var scope = new TempLinksTestScope(useSequences: true))
209     {
210         var links = scope.Links;
211         var sequences = scope.Sequences;
212
213         var sequence = new ulong[sequenceLength];
214         for (var i = 0; i < sequenceLength; i++)
215         {
216             sequence[i] = links.Create();
217         }
218
219         var createResults = sequences.CreateAllVariants2(sequence);
220
221         //var createResultsStrings = createResults.Select(x => x + ": " +
222             ↳ sequences.FormatSequence(x)).ToList();
223         //Global.Trash = createResultsStrings;
224
225         var partialSequence = new ulong[sequenceLength - 2];
226
227         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
228
229         var sw1 = Stopwatch.StartNew();
230         var searchResults1 =
231             ↳ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
232
233         var sw2 = Stopwatch.StartNew();
234         var searchResults2 =
235             ↳ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
236
237         //var sw3 = Stopwatch.StartNew();
238         //var searchResults3 =
239             ↳ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
240
241         var sw4 = Stopwatch.StartNew();
242         var searchResults4 =
243             ↳ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
244
245         //Global.Trash = searchResults3;
246
247         //var searchResults1Strings = searchResults1.Select(x => x + ": " +
248             ↳ sequences.FormatSequence(x)).ToList();
249         //Global.Trash = searchResults1Strings;
250
251         var intersection1 = createResults.Intersect(searchResults1).ToList();
252         Assert.True(intersection1.Count == createResults.Length);
253
254         var intersection2 = createResults.Intersect(searchResults2).ToList();
255         Assert.True(intersection2.Count == createResults.Length);
256
257         var intersection4 = createResults.Intersect(searchResults4).ToList();
258         Assert.True(intersection4.Count == createResults.Length);
259
260         for (var i = 0; i < sequenceLength; i++)
261         {
262             links.Delete(sequence[i]);
263         }
264     }
265 }
266
267 [Fact]
268 public static void BalancedPartialVariantsSearchTest()
269 {

```

```

264     const long sequenceLength = 200;
265
266     using (var scope = new TempLinksTestScope(useSequences: true))
267     {
268         var links = scope.Links;
269         var sequences = scope.Sequences;
270
271         var sequence = new ulong[sequenceLength];
272         for (var i = 0; i < sequenceLength; i++)
273         {
274             sequence[i] = links.Create();
275         }
276
277         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
278
279         var balancedVariant = balancedVariantConverter.Convert(sequence);
280
281         var partialSequence = new ulong[sequenceLength - 2];
282
283         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
284
285         var sw1 = Stopwatch.StartNew();
286         var searchResults1 =
287             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
288
289         var sw2 = Stopwatch.StartNew();
290         var searchResults2 =
291             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
292
293         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
294
295         Assert.True(searchResults2.Count == 1 && balancedVariant ==
296             ↪ searchResults2.First());
297
298         for (var i = 0; i < sequenceLength; i++)
299         {
300             links.Delete(sequence[i]);
301         }
302     }
303
304     [Fact(Skip = "Correct implementation is pending")]
305     public static void PatternMatchTest()
306     {
307         var zeroOrMany = Sequences.Sequences.ZeroOrMany;
308
309         using (var scope = new TempLinksTestScope(useSequences: true))
310         {
311             var links = scope.Links;
312             var sequences = scope.Sequences;
313
314             var e1 = links.Create();
315             var e2 = links.Create();
316
317             var sequence = new[]
318             {
319                 e1, e2, e1, e2 // mama / papa
320             };
321
322             var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
323
324             var balancedVariant = balancedVariantConverter.Convert(sequence);
325
326             // 1: [1]
327             // 2: [2]
328             // 3: [1,2]
329             // 4: [1,2,1,2]
330
331             var doublet = links.GetSource(balancedVariant);
332
333             var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
334
335             Assert.True(matchedSequences1.Count == 0);
336
337             var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
338
339             Assert.True(matchedSequences2.Count == 0);
340
341             var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);

```

```

341     Assert.True(matchedSequences3.Count == 0);
342
343     var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
344
345     Assert.Contains(doublet, matchedSequences4);
346     Assert.Contains(balancedVariant, matchedSequences4);
347
348     for (var i = 0; i < sequence.Length; i++)
349     {
350         links.Delete(sequence[i]);
351     }
352 }
353
354 [Fact]
355 public static void IndexTest()
356 {
357     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
358         ↪ true }, useSequences: true))
359     {
360         var links = scope.Links;
361         var sequences = scope.Sequences;
362         var indexer = sequences.Options.Indexer;
363
364         var e1 = links.Create();
365         var e2 = links.Create();
366
367         var sequence = new[]
368         {
369             e1, e2, e1, e2 // mama / papa
370         };
371
372         Assert.False(indexer.Index(sequence));
373
374         Assert.True(indexer.Index(sequence));
375     }
376 }
377
378 /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
379 ↪ D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
380 ↪ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
381 private static readonly string _exampleText =
382     @"([english
383     ↪ version] (https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

```

Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов  
 ↪ (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там  
 ↪ где есть место для нового начала? Разве пустота это не характеристика пространства?  
 ↪ Пространство это то, что можно чем-то наполнить?

[[чёрное пространство, белое  
 ↪ пространство] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png  
 ↪ ""чёрное пространство, белое пространство"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png)

Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая  
 ↪ форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?

[[чёрное пространство, чёрная  
 ↪ точка] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png  
 ↪ ""чёрное пространство, чёрная  
 ↪ точка"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)

А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть  
 ↪ так? Инверсия? Отражение? Сумма?

[[белая точка, чёрная  
 ↪ точка] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png ""белая  
 ↪ точка, чёрная  
 ↪ точка"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)

А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет  
 ↪ если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?  
 ↪ Гранью? Разделителем? Единицей?

[[две белые точки, чёрная вертикальная  
 ↪ линия] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png ""две  
 ↪ белые точки, чёрная вертикальная  
 ↪ линия"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)

397  
398 Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся  
→ только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится  
→ замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что  
→ можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец?  
→ Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если  
→ у него нет размера? Будет ли круг точкой? Точка состоящая из точек?

399  
400 [![белая вертикальная линия, чёрный  
→ круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png "белая  
→ вертикальная линия, чёрный  
→ круг")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)

401  
402 Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может  
→ тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально?  
→ Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли  
→ элементарная единица смысла?

403  
404 [![белый круг, чёрная горизонтальная  
→ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png "белый  
→ круг, чёрная горизонтальная  
→ линия")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)

405  
406 Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла "соединить,  
→ связать", есть ещё и смысл направления "от начала к концу"? От предка к потомку? От  
→ родителя к ребёнку? От общего к частному?

407  
408 [![белая горизонтальная линия, чёрная горизонтальная  
→ стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png  
→ "белая горизонтальная линия, чёрная горизонтальная  
→ стрелка")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)

409  
410 Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она  
→ может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть  
→ граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два  
→ объекта, как бы это выглядело?

411  
412 [![белая связь, чёрная направленная  
→ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png "белая  
→ связь, чёрная направленная  
→ связь")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)

413  
414 Допустим у нас есть смысл "связать" и смысл "направления", много ли это нам даёт? Много ли  
→ вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если  
→ можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие?  
→ Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в  
→ его конечном состоянии, если конечно конец определён направлением?

415  
416 [![белая обычная и направленная связи, чёрная типизированная  
→ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png "белая  
→ обычная и направленная связи, чёрная типизированная  
→ связь")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)

417  
418 А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри?  
→ Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать  
→ сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?

419  
420 [![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная  
→ связь с рекурсивной внутренней  
→ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png  
→ "белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная  
→ типизированная связь с рекурсивной внутренней структурой")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)

421  
422 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом  
→ рекурсии или фрактала?

423  
424 [![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная  
→ типизированная связь с двойной рекурсивной внутренней  
→ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png  
→ "белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная  
→ типизированная связь с двойной рекурсивной внутренней структурой")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)

425  
426 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?  
→ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?

427



```

428  [![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности,
↳      чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://
↳      /raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png "белая обычная и
↳      направленная связи со структурой из 8 цветных элементов последовательности, чёрная
↳      типизированная связь со структурой из 8 цветных элементов последовательности")](https://raw
↳      .githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)
429
430  ...
431
432  [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anima
↳      tion-500.gif
↳      "анимация")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
↳      -animation-500.gif)";
433
434
435      private static readonly string _exampleLoremIpsumText =
436          @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↳              incididunt ut labore et dolore magna aliqua.
437  Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
↳      consequat.";
438
439  [Fact]
440  public static void CompressionTest()
441  {
442      using (var scope = new TempLinksTestScope(useSequences: true))
443      {
444          var links = scope.Links;
445          var sequences = scope.Sequences;
446
447          var e1 = links.Create();
448          var e2 = links.Create();
449
450          var sequence = new[]
451          {
452              e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
453          };
454
455          var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
456          var totalSequenceSymbolFrequencyCounter = new
↳              TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
457          var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
↳              totalSequenceSymbolFrequencyCounter);
458          var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
↳              balancedVariantConverter, doubletFrequenciesCache);
459
460          var compressedVariant = compressingConverter.Convert(sequence);
461
462          // 1: [1]          (1->1) point
463          // 2: [2]          (2->2) point
464          // 3: [1,2]        (1->2) doublet
465          // 4: [1,2,1,2]    (3->3) doublet
466
467          Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
468          Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
469          Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
470          Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
471
472          var source = _constants.SourcePart;
473          var target = _constants.TargetPart;
474
475          Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
476          Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
477          Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
478          Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
479
480          // 4 - length of sequence
481          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
↳              == sequence[0]);
482          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
↳              == sequence[1]);
483          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
↳              == sequence[2]);
484          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
↳              == sequence[3]);
485      }
486  }
487
488  [Fact]
489  public static void CompressionEfficiencyTest()

```

```

490 {
491     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
        ↳ StringSplitOptions.RemoveEmptyEntries);
492     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
493     var totalCharacters = arrays.Select(x => x.Length).Sum();
494
495     using (var scope1 = new TempLinksTestScope(useSequences: true))
496     using (var scope2 = new TempLinksTestScope(useSequences: true))
497     using (var scope3 = new TempLinksTestScope(useSequences: true))
498     {
499         scope1.Links.Unsync.UseUnicode();
500         scope2.Links.Unsync.UseUnicode();
501         scope3.Links.Unsync.UseUnicode();
502
503         var balancedVariantConverter1 = new
        ↳ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
504         var totalSequenceSymbolFrequencyCounter = new
        ↳ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
505         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
        ↳ totalSequenceSymbolFrequencyCounter);
506         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
        ↳ balancedVariantConverter1, linkFrequenciesCache1,
        ↳ doInitialFrequenciesIncrement: false);
507
508         var compressor2 = scope2.Sequences;
509         var compressor3 = scope3.Sequences;
510
511         var constants = Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
512
513         var sequences = compressor3;
514         //var meaningRoot = links.CreatePoint();
515         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
516         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
517         //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
        ↳ constants.Itself);
518
519         //var unaryNumberToAddressConveter = new
        ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
520         //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
        ↳ unaryOne);
521         //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
        ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
522         //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
        ↳ frequencyPropertyMarker, frequencyMarker);
523         //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
        ↳ frequencyPropertyOperator, frequencyIncrementer);
524         //var linkToItsFrequencyNumberConverter = new
        ↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
        ↳ unaryNumberToAddressConveter);
525
526         var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
        ↳ totalSequenceSymbolFrequencyCounter);
527
528         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
        ↳ ncyNumberConverter<ulong>(linkFrequenciesCache3);
529
530         var sequenceToItsLocalElementLevelsConverter = new
        ↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
        ↳ linkToItsFrequencyNumberConverter);
531         var optimalVariantConverter = new
        ↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
        ↳ sequenceToItsLocalElementLevelsConverter);
532
533         var compressed1 = new ulong[arrays.Length];
534         var compressed2 = new ulong[arrays.Length];
535         var compressed3 = new ulong[arrays.Length];
536
537         var START = 0;
538         var END = arrays.Length;
539
540         //for (int i = START; i < END; i++)
541         //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
542
543         var initialCount1 = scope2.Links.Unsync.Count();
544
545         var sw1 = Stopwatch.StartNew();
546
547         for (int i = START; i < END; i++)

```

```

548     {
549         linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
550         compressed1[i] = compressor1.Convert(arrays[i]);
551     }
552
553     var elapsed1 = sw1.Elapsed;
554
555     var balancedVariantConverter2 = new
556     ↪ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
557
558     var initialCount2 = scope2.Links.Unsync.Count();
559
560     var sw2 = Stopwatch.StartNew();
561
562     for (int i = START; i < END; i++)
563     {
564         compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
565     }
566
567     var elapsed2 = sw2.Elapsed;
568
569     for (int i = START; i < END; i++)
570     {
571         linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
572     }
573
574     var initialCount3 = scope3.Links.Unsync.Count();
575
576     var sw3 = Stopwatch.StartNew();
577
578     for (int i = START; i < END; i++)
579     {
580         //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
581         compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
582     }
583
584     var elapsed3 = sw3.Elapsed;
585
586     Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
587     ↪ Optimal variant: {elapsed3}");
588
589     // Assert.True(elapsed1 > elapsed2);
590
591     // Checks
592     for (int i = START; i < END; i++)
593     {
594         var sequence1 = compressed1[i];
595         var sequence2 = compressed2[i];
596         var sequence3 = compressed3[i];
597
598         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
599         ↪ scope1.Links.Unsync);
600
601         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
602         ↪ scope2.Links.Unsync);
603
604         var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
605         ↪ scope3.Links.Unsync);
606
607         var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
608         ↪ link.IsPartialPoint());
609         var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
610         ↪ link.IsPartialPoint());
611         var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
612         ↪ link.IsPartialPoint());
613
614         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
615         ↪ arrays[i].Length > 3)
616         //    Assert.False(structure1 == structure2);
617         //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
618         ↪ arrays[i].Length > 3)
619         //    Assert.False(structure3 == structure2);
620
621         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
622         Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
623     }
624
625     Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
626     ↪ totalCharacters);

```

```

616 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
    ↳ totalCharacters);
617 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
    ↳ totalCharacters);
618
619 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
    ↳ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
    ↳ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
    ↳ totalCharacters}");
620
621 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
    ↳ scope2.Links.Unsync.Count() - initialCount2);
622 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
    ↳ scope2.Links.Unsync.Count() - initialCount2);
623
624 var duplicateProvider1 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
625 var duplicateProvider2 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
626 var duplicateProvider3 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
627
628 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
629 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
630 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
631
632 var duplicates1 = duplicateCounter1.Count();
633
634 ConsoleHelpers.Debug("-----");
635
636 var duplicates2 = duplicateCounter2.Count();
637
638 ConsoleHelpers.Debug("-----");
639
640 var duplicates3 = duplicateCounter3.Count();
641
642 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
643
644 linkFrequenciesCache1.ValidateFrequencies();
645 linkFrequenciesCache3.ValidateFrequencies();
646 }
647 }
648
649 [Fact]
650 public static void CompressionStabilityTest()
651 {
652     // TODO: Fix bug (do a separate test)
653     //const ulong minNumbers = 0;
654     //const ulong maxNumbers = 1000;
655
656     const ulong minNumbers = 10000;
657     const ulong maxNumbers = 12500;
658
659     var strings = new List<string>();
660
661     for (ulong i = minNumbers; i < maxNumbers; i++)
662     {
663         strings.Add(i.ToString());
664     }
665
666     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
667     var totalCharacters = arrays.Select(x => x.Length).Sum();
668
669     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
    ↳ SequencesOptions<ulong> { UseCompression = true,
    ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
    using (var scope2 = new TempLinksTestScope(useSequences: true))
    {
670         scope1.Links.UseUnicode();
671         scope2.Links.UseUnicode();
672
673         //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
674         var compressor1 = scope1.Sequences;
675         var compressor2 = scope2.Sequences;
676
677         var compressed1 = new ulong[arrays.Length];
678         var compressed2 = new ulong[arrays.Length];
679
680         var sw1 = Stopwatch.StartNew();

```

```

683
684 var START = 0;
685 var END = arrays.Length;
686
687 // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
688 // Stability issue starts at 10001 or 11000
689 //for (int i = START; i < END; i++)
690 //{
691 //    var first = compressor1.Compress(arrays[i]);
692 //    var second = compressor1.Compress(arrays[i]);
693
694 //    if (first == second)
695 //        compressed1[i] = first;
696 //    else
697 //    {
698 //        // TODO: Find a solution for this case
699 //    }
700 //}
701
702 for (int i = START; i < END; i++)
703 {
704     var first = compressor1.Create(arrays[i]);
705     var second = compressor1.Create(arrays[i]);
706
707     if (first == second)
708     {
709         compressed1[i] = first;
710     }
711     else
712     {
713         // TODO: Find a solution for this case
714     }
715 }
716
717 var elapsed1 = sw1.Elapsed;
718
719 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
720
721 var sw2 = Stopwatch.StartNew();
722
723 for (int i = START; i < END; i++)
724 {
725     var first = balancedVariantConverter.Convert(arrays[i]);
726     var second = balancedVariantConverter.Convert(arrays[i]);
727
728     if (first == second)
729     {
730         compressed2[i] = first;
731     }
732 }
733
734 var elapsed2 = sw2.Elapsed;
735
736 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
737 ↪ {elapsed2}");
738
739 Assert.True(elapsed1 > elapsed2);
740
741 // Checks
742 for (int i = START; i < END; i++)
743 {
744     var sequence1 = compressed1[i];
745     var sequence2 = compressed2[i];
746
747     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
748     {
749         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
750 ↪ scope1.Links);
751
752         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
753 ↪ scope2.Links);
754
755         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
756 ↪ link.IsPartialPoint());
757         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
758 ↪ link.IsPartialPoint());
759
760         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
761 ↪ arrays[i].Length > 3)

```

```

756         // Assert.False(structure1 == structure2);
757
758         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
759     }
760 }
761
762 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
763 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
764
765 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
    ↳ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
    ↳ totalCharacters}");
766
767 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
768
769 //compressor1.ValidateFrequencies();
770 }
771 }
772
773 [Fact]
774 public static void RandomNumbersCompressionQualityTest()
775 {
776     const ulong N = 500;
777
778     //const ulong minNumbers = 10000;
779     //const ulong maxNumbers = 20000;
780
781     //var strings = new List<string>();
782
783     //for (ulong i = 0; i < N; i++)
784     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
785     ↳ maxNumbers).ToString());
786
787     var strings = new List<string>();
788
789     for (ulong i = 0; i < N; i++)
790     {
791         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
792     }
793
794     strings = strings.Distinct().ToList();
795
796     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
797     var totalCharacters = arrays.Select(x => x.Length).Sum();
798
799     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
800     ↳ SequencesOptions<ulong> { UseCompression = true,
801     ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
802     using (var scope2 = new TempLinksTestScope(useSequences: true))
803     {
804         scope1.Links.UseUnicode();
805         scope2.Links.UseUnicode();
806
807         var compressor1 = scope1.Sequences;
808         var compressor2 = scope2.Sequences;
809
810         var compressed1 = new ulong[arrays.Length];
811         var compressed2 = new ulong[arrays.Length];
812
813         var sw1 = Stopwatch.StartNew();
814
815         var START = 0;
816         var END = arrays.Length;
817
818         for (int i = START; i < END; i++)
819         {
820             compressed1[i] = compressor1.Create(arrays[i]);
821         }
822
823         var elapsed1 = sw1.Elapsed;
824
825         var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
826
827         var sw2 = Stopwatch.StartNew();
828
829         for (int i = START; i < END; i++)
830         {
831             compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
832         }
833     }

```

```

831     var elapsed2 = sw2.Elapsed;
832
833     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
834         ↳ {elapsed2}");
835
836     Assert.True(elapsed1 > elapsed2);
837
838     // Checks
839     for (int i = START; i < END; i++)
840     {
841         var sequence1 = compressed1[i];
842         var sequence2 = compressed2[i];
843
844         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
845         {
846             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
847                 ↳ scope1.Links);
848
849             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
850                 ↳ scope2.Links);
851
852             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
853         }
854     }
855
856     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
857     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
858
859     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
860         ↳ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
861         ↳ totalCharacters}}");
862
863     // Can be worse than balanced variant
864     //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
865
866     //compressor1.ValidateFrequencies();
867 }
868
869 [Fact]
870 public static void AllTreeBreakDownAtSequencesCreationBugTest()
871 {
872     // Made out of AllPossibleConnectionsTest test.
873
874     //const long sequenceLength = 5; //100% bug
875     const long sequenceLength = 4; //100% bug
876     //const long sequenceLength = 3; //100% _no_bug (ok)
877
878     using (var scope = new TempLinksTestScope(useSequences: true))
879     {
880         var links = scope.Links;
881         var sequences = scope.Sequences;
882
883         var sequence = new ulong[sequenceLength];
884         for (var i = 0; i < sequenceLength; i++)
885         {
886             sequence[i] = links.Create();
887         }
888
889         var createResults = sequences.CreateAllVariants2(sequence);
890
891         Global.Trash = createResults;
892
893         for (var i = 0; i < sequenceLength; i++)
894         {
895             links.Delete(sequence[i]);
896         }
897     }
898 }
899
900 [Fact]
901 public static void AllPossibleConnectionsTest()
902 {
903     const long sequenceLength = 5;
904
905     using (var scope = new TempLinksTestScope(useSequences: true))
906     {
907         var links = scope.Links;
908         var sequences = scope.Sequences;

```

```

905
906     var sequence = new ulong[sequenceLength];
907     for (var i = 0; i < sequenceLength; i++)
908     {
909         sequence[i] = links.Create();
910     }
911
912     var createResults = sequences.CreateAllVariants2(sequence);
913     var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
914
915     for (var i = 0; i < 1; i++)
916     {
917         var sw1 = Stopwatch.StartNew();
918         var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
919
920         var sw2 = Stopwatch.StartNew();
921         var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
922
923         var sw3 = Stopwatch.StartNew();
924         var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
925
926         var sw4 = Stopwatch.StartNew();
927         var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
928
929         Global.Trash = searchResults3;
930         Global.Trash = searchResults4; //-V3008
931
932         var intersection1 = createResults.Intersect(searchResults1).ToList();
933         Assert.True(intersection1.Count == createResults.Length);
934
935         var intersection2 = reverseResults.Intersect(searchResults1).ToList();
936         Assert.True(intersection2.Count == reverseResults.Length);
937
938         var intersection0 = searchResults1.Intersect(searchResults2).ToList();
939         Assert.True(intersection0.Count == searchResults2.Count);
940
941         var intersection3 = searchResults2.Intersect(searchResults3).ToList();
942         Assert.True(intersection3.Count == searchResults3.Count);
943
944         var intersection4 = searchResults3.Intersect(searchResults4).ToList();
945         Assert.True(intersection4.Count == searchResults4.Count);
946     }
947
948     for (var i = 0; i < sequenceLength; i++)
949     {
950         links.Delete(sequence[i]);
951     }
952 }
953
954 [Fact(Skip = "Correct implementation is pending")]
955 public static void CalculateAllUsagesTest()
956 {
957     const long sequenceLength = 3;
958
959     using (var scope = new TempLinksTestScope(useSequences: true))
960     {
961         var links = scope.Links;
962         var sequences = scope.Sequences;
963
964         var sequence = new ulong[sequenceLength];
965         for (var i = 0; i < sequenceLength; i++)
966         {
967             sequence[i] = links.Create();
968         }
969
970         var createResults = sequences.CreateAllVariants2(sequence);
971
972         //var reverseResults =
973         ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
974
975         for (var i = 0; i < 1; i++)
976         {
977             var linksTotalUsages1 = new ulong[links.Count() + 1];
978
979             sequences.CalculateAllUsages(linksTotalUsages1);
980
981             var linksTotalUsages2 = new ulong[links.Count() + 1];
982
983             sequences.CalculateAllUsages2(linksTotalUsages2);

```



```

984         var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
985         Assert.True(intersection1.Count == linksTotalUsages2.Length);
986     }
987 }
988
989 for (var i = 0; i < sequenceLength; i++)
990 {
991     links.Delete(sequence[i]);
992 }
993 }
994 }
995 }
996 }

```

./Platform.Data.Doublets.Tests/TempLinksTestScope.cs

```

1  using System.IO;
2  using Platform.Disposables;
3  using Platform.Data.Doublets.ResizableDirectMemory;
4  using Platform.Data.Doublets.Sequences;
5  using Platform.Data.Doublets.Decorators;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public class TempLinksTestScope : DisposableBase
10     {
11         public readonly ILinks<ulong> MemoryAdapter;
12         public readonly SynchronizedLinks<ulong> Links;
13         public readonly Sequences.Sequences Sequences;
14         public readonly string TempFilename;
15         public readonly string TempTransactionLogFilename;
16         private readonly bool _deleteFiles;
17
18         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
19             ↪ useLog = false)
20             : this(new SequencesOptions<ulong>(), deleteFiles, useSequences, useLog)
21         {
22         }
23
24         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
25             ↪ true, bool useSequences = false, bool useLog = false)
26         {
27             _deleteFiles = deleteFiles;
28             TempFilename = Path.GetTempFileName();
29             TempTransactionLogFilename = Path.GetTempFileName();
30
31             var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
32
33             MemoryAdapter = useLog ? (ILinks<ulong>)new
34                 ↪ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
35                 ↪ coreMemoryAdapter;
36
37             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
38             if (useSequences)
39             {
40                 Sequences = new Sequences.Sequences(Links, sequencesOptions);
41             }
42         }
43
44         protected override void Dispose(bool manual, bool wasDisposed)
45         {
46             if (!wasDisposed)
47             {
48                 Links.Unsync.DisposeIfPossible();
49                 if (_deleteFiles)
50                 {
51                     DeleteFiles();
52                 }
53             }
54         }
55
56         public void DeleteFiles()
57         {
58             File.Delete(TempFilename);
59             File.Delete(TempTransactionLogFilename);
60         }
61     }
62 }

```

./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs

```
1 using Xunit;
2 using Platform.Random;
3 using Platform.Data.Doublets.Converters;
4
5 namespace Platform.Data.Doublets.Tests
6 {
7     public static class UnaryNumberConvertersTests
8     {
9         [Fact]
10        public static void ConvertersTest()
11        {
12            using (var scope = new TempLinksTestScope())
13            {
14                const int N = 10;
15                var links = scope.Links;
16                var meaningRoot = links.CreatePoint();
17                var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
18                var powerOf2ToUnaryNumberConverter = new
19                    ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
20                var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
21                    ↪ powerOf2ToUnaryNumberConverter);
22                var random = new System.Random(0);
23                ulong[] numbers = new ulong[N];
24                ulong[] unaryNumbers = new ulong[N];
25                for (int i = 0; i < N; i++)
26                {
27                    numbers[i] = random.NextUInt64();
28                    unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
29                }
30                var fromUnaryNumberConverterUsingOrOperation = new
31                    ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
32                    ↪ powerOf2ToUnaryNumberConverter);
33                var fromUnaryNumberConverterUsingAddOperation = new
34                    ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
35                for (int i = 0; i < N; i++)
36                {
37                    Assert.Equal(numbers[i],
38                        ↪ fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
39                    Assert.Equal(numbers[i],
40                        ↪ fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
41                }
42            }
43        }
44    }
45 }
```

## Index

./Platform.Data.Doublets.Tests/ComparisonTests.cs, 132  
./Platform.Data.Doublets.Tests/DoubletLinksTests.cs, 133  
./Platform.Data.Doublets.Tests/EqualityTests.cs, 136  
./Platform.Data.Doublets.Tests/LinksTests.cs, 137  
./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 150  
./Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 152  
./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 153  
./Platform.Data.Doublets.Tests/ScopeTests.cs, 153  
./Platform.Data.Doublets.Tests/SequencesTests.cs, 154  
./Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 169  
./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 169  
./Platform.Data.Doublets/Converters/AddressToUnaryNumberConverter.cs, 1  
./Platform.Data.Doublets/Converters/LinkToltsFrequencyNumberConveter.cs, 1  
./Platform.Data.Doublets/Converters/PowerOf2ToUnaryNumberConverter.cs, 2  
./Platform.Data.Doublets/Converters/UnaryNumberToAddressAddOperationConverter.cs, 2  
./Platform.Data.Doublets/Converters/UnaryNumberToAddressOrOperationConverter.cs, 3  
./Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs, 4  
./Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs, 4  
./Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs, 4  
./Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs, 5  
./Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs, 5  
./Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs, 6  
./Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs, 6  
./Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs, 6  
./Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs, 7  
./Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs, 7  
./Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs, 7  
./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs, 8  
./Platform.Data.Doublets/Decorators/UInt64Links.cs, 8  
./Platform.Data.Doublets/Decorators/UniLinks.cs, 9  
./Platform.Data.Doublets/Doublet.cs, 14  
./Platform.Data.Doublets/DoubletComparer.cs, 14  
./Platform.Data.Doublets/Hybrid.cs, 14  
./Platform.Data.Doublets/ILinks.cs, 15  
./Platform.Data.Doublets/ILinksExtensions.cs, 16  
./Platform.Data.Doublets/ISynchronizedLinks.cs, 28  
./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs, 26  
./Platform.Data.Doublets/Incrementers/LinkFrequencyIncrementer.cs, 27  
./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs, 27  
./Platform.Data.Doublets/Link.cs, 28  
./Platform.Data.Doublets/LinkExtensions.cs, 30  
./Platform.Data.Doublets/LinksOperatorBase.cs, 30  
./Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 30  
./Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 31  
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs, 40  
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs, 41  
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 32  
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs, 54  
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs, 55  
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 47  
./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 61  
./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 62  
./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 65  
./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 65  
./Platform.Data.Doublets/Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 66  
./Platform.Data.Doublets/Sequences/CriteriaMatchers/DefaultSequenceElementCriterionMatcher.cs, 67  
./Platform.Data.Doublets/Sequences/CriteriaMatchers/MarkedSequenceCriterionMatcher.cs, 67  
./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 67  
./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 68  
./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 68  
./Platform.Data.Doublets/Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs, 70  
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 71  
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 72  
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 73  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 73  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 73

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 74  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 74  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 75  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 75  
./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 76  
./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 76  
./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 77  
./Platform.Data.Doublets/Sequences/Sequences.Experiments.ReadSequence.cs, 112  
./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 86  
./Platform.Data.Doublets/Sequences/Sequences.cs, 77  
./Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 114  
./Platform.Data.Doublets/Sequences/SequencesIndexer.cs, 114  
./Platform.Data.Doublets/Sequences/SequencesOptions.cs, 115  
./Platform.Data.Doublets/Sequences/UnicodeMap.cs, 117  
./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 119  
./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 119  
./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 120  
./Platform.Data.Doublets/Stacks/Stack.cs, 121  
./Platform.Data.Doublets/Stacks/StackExtensions.cs, 121  
./Platform.Data.Doublets/SynchronizedLinks.cs, 122  
./Platform.Data.Doublets/UInt64Link.cs, 122  
./Platform.Data.Doublets/UInt64LinkExtensions.cs, 124  
./Platform.Data.Doublets/UInt64LinksExtensions.cs, 125  
./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 127