LinksPlatform's Platform.Data.Doublets Class Library

./Converters/AddressToUnaryNumberConverter.cs

```csharp
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Reflection;
using Platform.Numbers;

namespace Platform.Data.Doublets.Converters
{
    public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
        IConverter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;

        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;

        public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
            powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
            powerOf2ToUnaryNumberConverter;

        public TLink Convert(TLink sourceAddress)
        {
            var number = sourceAddress;
            var target = Links.Constants.Null;
            for (int i = 0; i < CachedTypeInfo<TLink>.BitsLength; i++)
            {
                if (_equalityComparer.Equals(ArithmeticHelpers.And(number, Integer<TLink>.One),
                    Integer<TLink>.One))
                {
                    target = _equalityComparer.Equals(target, Links.Constants.Null)
                        ? _powerOf2ToUnaryNumberConverter.Convert(i)
                        : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
                }
                number = (Integer<TLink>)((ulong)(Integer<TLink>)number >> 1); // Should be
                    BitwiseHelpers.ShiftRight(number, 1);
                if (_equalityComparer.Equals(number, default))
                {
                    break;
                }
            }
            return target;
        }
    }
}
```

./Converters/LinkToItsFrequencyNumberConveter.cs

```csharp
using System;
using System.Collections.Generic;
using Platform.Interfaces;

namespace Platform.Data.Doublets.Converters
{
    public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
        IConverter<Doublet<TLink>, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;

        private readonly ISpecificPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;

        public LinkToItsFrequencyNumberConveter(
            ILinks<TLink> links,
            ISpecificPropertyOperator<TLink, TLink> frequencyPropertyOperator,
            IConverter<TLink> unaryNumberToAddressConverter)
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
        }

        public TLink Convert(Doublet<TLink> doublet)
        {
            var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
            if (_equalityComparer.Equals(link, Links.Constants.Null))
            {
                throw new ArgumentException($"Link with {doublet.Source} source and
                    {doublet.Target} target not found.", nameof(doublet));
            }
```

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

./Converters/PowerOf2ToUnaryNumberConverter.cs

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

./Converters/UnaryNumberToAddressAddOperationConverter.cs

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

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

./Converters/UnaryNumberToAddressOrOperationConverter.cs

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

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

./Decorators/LinksCascadeDependenciesResolver.cs

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

./Decorators/LinksCascadeUniquenessAndDependenciesResolver.cs

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

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

./Decorators/LinksDecoratorBase.cs

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

./Decorators/LinksDependenciesValidator.cs

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

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

./Decorators/LinksDisposableDecoratorBase.cs

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

./Decorators/LinksInnerReferenceValidator.cs

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

```
36          }
37      }
```

./Decorators/LinksNonExistentReferencesCreator.cs

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

./Decorators/LinksNullToSelfReferenceResolver.cs

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

./Decorators/LinksSelfReferenceResolver.cs

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

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

./Decorators/LinksUniquenessResolver.cs

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

./Decorators/LinksUniquenessValidator.cs

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

./Decorators/NonNullContentsLinkDeletionResolver.cs

```csharp
namespace Platform.Data.Doublets.Decorators
{
    public class NonNullContentsLinkDeletionResolver<T> : LinksDecoratorBase<T>
    {
        public NonNullContentsLinkDeletionResolver(ILinks<T> links) : base(links) { }

        public override void Delete(T link)
        {
            Links.Update(link, Constants.Null, Constants.Null);
            base.Delete(link);
        }
    }
}
```

./Decorators/UInt64Links.cs

```csharp
using System;
using System.Collections.Generic;
using Platform.Collections;
using Platform.Collections.Arrays;

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// Представляет объект для работы с базой данных (файлом) в формате Links (массива
    ///  взаимосвязей).
    /// </summary>
    /// <remarks>
    /// Возможные оптимизации:
    /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
    ///     + меньше объём БД
    ///     - меньше производительность
    ///     - больше ограничение на количество связей в БД)
    /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
    ///     + меньше объём БД
    ///     - больше сложность
    ///
    ///     AVL - высота дерева может позволить точно расчитать размер дерева, нет необходимости
    ///  в SBT.
    ///     AVL дерево можно прошить.
    ///
    /// Текущее теоретическое ограничение на размер связей - long.MaxValue
    /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
    ///  (битовыми строками) - вариант матрицы (выстраеваемой лениво).
    ///
    /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
    ///  выбрасываться только при #if DEBUG
    /// </remarks>
    public class UInt64Links : LinksDisposableDecoratorBase<ulong>
    {
        public UInt64Links(ILinks<ulong> links) : base(links) { }

        public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
        {
            this.EnsureLinkIsAnyOrExists(restrictions);
            return Links.Each(handler, restrictions);
        }

        public override ulong Create() => Links.CreatePoint();

        public override ulong Update(IList<ulong> restrictions)
        {
            if (restrictions.IsNullOrEmpty())
            {
                return Constants.Null;
            }
            // TODO: Remove usages of these hacks (these should not be backwards compatible)
            if (restrictions.Count == 2)
            {
                return this.Merge(restrictions[0], restrictions[1]);
            }
            if (restrictions.Count == 4)
            {
                return this.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
                 restrictions[2], restrictions[3]);
            }
            // TODO: Looks like this is a common type of exceptions linked with restrictions
            //  support
            if (restrictions.Count != 3)
```

```csharp
58              {
59                  throw new NotSupportedException();
60              }
61              var updatedLink = restrictions[Constants.IndexPart];
62              this.EnsureLinkExists(updatedLink, nameof(Constants.IndexPart));
63              var newSource = restrictions[Constants.SourcePart];
64              this.EnsureLinkIsItselfOrExists(newSource, nameof(Constants.SourcePart));
65              var newTarget = restrictions[Constants.TargetPart];
66              this.EnsureLinkIsItselfOrExists(newTarget, nameof(Constants.TargetPart));
67              var existedLink = Constants.Null;
68              if (newSource != Constants.Itself && newTarget != Constants.Itself)
69              {
70                  existedLink = this.SearchOrDefault(newSource, newTarget);
71              }
72              if (existedLink == Constants.Null)
73              {
74                  var before = Links.GetLink(updatedLink);
75                  if (before[Constants.SourcePart] != newSource || before[Constants.TargetPart] !=
   ↪   newTarget)
76                  {
77                      Links.Update(updatedLink, newSource == Constants.Itself ? updatedLink :
   ↪   newSource,
78                                                  newTarget == Constants.Itself ? updatedLink :
   ↪   newTarget);
79                  }
80                  return updatedLink;
81              }
82              else
83              {
84                  // Replace one link with another (replaced link is deleted, children are updated
   ↪   or deleted), it is actually merge operation
85                  return this.Merge(updatedLink, existedLink);
86              }
87          }
88
89          /// <summary>Удаляет связь с указанным индексом.</summary>
90          /// <param name="link">Индекс удаляемой связи.</param>
91          public override void Delete(ulong link)
92          {
93              this.EnsureLinkExists(link);
94              Links.Update(link, Constants.Null, Constants.Null);
95              var referencesCount = Links.Count(Constants.Any, link);
96              if (referencesCount > 0)
97              {
98                  var references = new ulong[referencesCount];
99                  var referencesFiller = new ArrayFiller<ulong, ulong>(references,
   ↪   Constants.Continue);
100                 Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, link);
101                 //references.Sort(); // TODO: Решить необходимо ли для корректного порядка
   ↪   отмены операций в транзакциях
102                 for (var i = (long)referencesCount - 1; i >= 0; i--)
103                 {
104                     if (this.Exists(references[i]))
105                     {
106                         Delete(references[i]);
107                     }
108                 }
109                 //else
110                 // TODO: Определить почему здесь есть связи, которых не существует
111             }
112             Links.Delete(link);
113         }
114     }
115 }
```

./Decorators/UniLinks.cs

```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Platform.Collections;
5  using Platform.Collections.Arrays;
6  using Platform.Collections.Lists;
7  using Platform.Helpers.Scopes;
8  using Platform.Data.Constants;
9  using Platform.Data.Universal;
10 using System.Collections.ObjectModel;
11
12 namespace Platform.Data.Doublets.Decorators
13 {
```

```csharp
        /// <remarks>
        /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
        /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed
        ///   by itself. But can cause creation (update from nothing) or deletion (update to nothing).
        /// 
        /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
        ///   DefaultUniLinksBase, that contains logic itself and can be implemented using both
        ///   IDoubletLinks and ILinks.)
        /// </remarks>
        internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
        {
            private static readonly EqualityComparer<TLink> _equalityComparer =
              EqualityComparer<TLink>.Default;

            public UniLinks(ILinks<TLink> links) : base(links) { }

            private struct Transition
            {
                public IList<TLink> Before;
                public IList<TLink> After;

                public Transition(IList<TLink> before, IList<TLink> after)
                {
                    Before = before;
                    After = after;
                }
            }

            public static readonly TLink NullConstant = Use<LinksCombinedConstants<TLink, TLink,
              int>>.Single.Null;
            public static readonly IReadOnlyList<TLink> NullLink = new ReadOnlyCollection<TLink>(new
              List<TLink> { NullConstant, NullConstant, NullConstant });

            // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
            //   (Links-Expression)
            public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
              matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
              substitutedHandler)
            {
                ////List<Transition> transitions = null;
                ////if (!restriction.IsNullOrEmpty())
                ////{
                ////    // Есть причина делать проход (чтение)
                ////    if (matchedHandler != null)
                ////    {
                ////        if (!substitution.IsNullOrEmpty())
                ////        {
                ////            // restriction => { 0, 0, 0 } | { 0 } // Create
                ////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
                ////   Create / Update
                ////            // substitution => { 0, 0, 0 } | { 0 } // Delete
                ////            transitions = new List<Transition>();
                ////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
                ////            {
                ////                // If index is Null, that means we always ignore every other
                ////   value (they are also Null by definition)
                ////                var matchDecision = matchedHandler(, NullLink);
                ////                if (Equals(matchDecision, Constants.Break))
                ////                    return false;
                ////                if (!Equals(matchDecision, Constants.Skip))
                ////                    transitions.Add(new Transition(matchedLink, newValue));
                ////            }
                ////            else
                ////            {
                ////                Func<T, bool> handler;
                ////                handler = link =>
                ////                {
                ////                    var matchedLink = Memory.GetLinkValue(link);
                ////                    var newValue = Memory.GetLinkValue(link);
                ////                    newValue[Constants.IndexPart] = Constants.Itself;
                ////                    newValue[Constants.SourcePart] =
                ////   Equals(substitution[Constants.SourcePart], Constants.Itself) ?
                ////   matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
                ////                    newValue[Constants.TargetPart] =
                ////   Equals(substitution[Constants.TargetPart], Constants.Itself) ?
                ////   matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
                ////                    var matchDecision = matchedHandler(matchedLink, newValue);
                ////                    if (Equals(matchDecision, Constants.Break))
```

```csharp
////                            return false;
////                        if (!Equals(matchDecision, Constants.Skip))
////                            transitions.Add(new Transition(matchedLink, newValue));
////                        return true;
////                    };
////                    if (!Memory.Each(handler, restriction))
////                        return Constants.Break;
////                }
////            }
////            else
////            {
////                Func<T, bool> handler = link =>
////                {
////                    var matchedLink = Memory.GetLinkValue(link);
////                    var matchDecision = matchedHandler(matchedLink, matchedLink);
////                    return !Equals(matchDecision, Constants.Break);
////                };
////                if (!Memory.Each(handler, restriction))
////                    return Constants.Break;
////            }
////        }
////        else
////        {
////            if (substitution != null)
////            {
////                transitions = new List<IList<T>>();
////                Func<T, bool> handler = link =>
////                {
////                    var matchedLink = Memory.GetLinkValue(link);
////                    transitions.Add(matchedLink);
////                    return true;
////                };
////                if (!Memory.Each(handler, restriction))
////                    return Constants.Break;
////            }
////            else
////            {
////                return Constants.Continue;
////            }
////        }
////}
////if (substitution != null)
////{
////    // Есть причина делать замену (запись)
////    if (substitutedHandler != null)
////    {
////    }
////    else
////    {
////    }
////}
////return Constants.Continue;

//if (restriction.IsNullOrEmpty()) // Create
//{
//    substitution[Constants.IndexPart] = Memory.AllocateLink();
//    Memory.SetLinkValue(substitution);
//}
//else if (substitution.IsNullOrEmpty()) // Delete
//{
//    Memory.FreeLink(restriction[Constants.IndexPart]);
//}
//else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
//{
//    // No need to collect links to list
//    // Skip == Continue
//    // No need to check substituedHandler
//    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
//        Constants.Break), restriction))
//        return Constants.Break;
//}
//else // Update
//{
//    //List<IList<T>> matchedLinks = null;
//    if (matchedHandler != null)
//    {
//        matchedLinks = new List<IList<T>>();
//        Func<T, bool> handler = link =>
```

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

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

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

./DoubletComparer.cs

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

./Doublet.cs

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

```csharp
            }

            public override string ToString() => $"{Source}->{Target}";

            public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
                ↪  && _equalityComparer.Equals(Target, other.Target);
        }
    }
```

./Hybrid.cs
```csharp
using System;
using System.Reflection;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;

namespace Platform.Data.Doublets
{
    public class Hybrid<T>
    {
        public readonly T Value;
        public bool IsNothing => Convert.ToInt64(To.Signed(Value)) == 0;
        public bool IsInternal => Convert.ToInt64(To.Signed(Value)) > 0;
        public bool IsExternal => Convert.ToInt64(To.Signed(Value)) < 0;
        public long AbsoluteValue => Math.Abs(Convert.ToInt64(To.Signed(Value)));

        public Hybrid(T value)
        {
            if (CachedTypeInfo<T>.IsSigned)
            {
                throw new NotSupportedException();
            }
            Value = value;
        }

        public Hybrid(object value) => Value = To.UnsignedAs<T>(Convert.ChangeType(value,
            ↪  CachedTypeInfo<T>.SignedVersion));

        public Hybrid(object value, bool isExternal)
        {
            var signedType = CachedTypeInfo<T>.SignedVersion;
            var signedValue = Convert.ChangeType(value, signedType);
            var abs =
                ↪  typeof(MathHelpers).GetTypeInfo().GetMethod("Abs").MakeGenericMethod(signedType);
            var negate = typeof(MathHelpers).GetTypeInfo().GetMethod("Negate").MakeGenericMethod
                ↪  (signedType);
            var absoluteValue = abs.Invoke(null, new[] { signedValue });
            var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
                ↪  absoluteValue;
            Value = To.UnsignedAs<T>(resultValue);
        }

        public static implicit operator Hybrid<T>(T integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(ulong integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(long integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(uint integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(int integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(ushort integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(short integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(byte integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(sbyte integer) => new Hybrid<T>(integer);

        public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;

        public static explicit operator ulong(Hybrid<T> hybrid) =>
            ↪  Convert.ToUInt64(hybrid.Value);

        public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;

        public static explicit operator uint(Hybrid<T> hybrid) => Convert.ToUInt32(hybrid.Value);
```

```csharp
            public static explicit operator int(Hybrid<T> hybrid) =>
                Convert.ToInt32(hybrid.AbsoluteValue);

            public static explicit operator ushort(Hybrid<T> hybrid) =>
                Convert.ToUInt16(hybrid.Value);

            public static explicit operator short(Hybrid<T> hybrid) =>
                Convert.ToInt16(hybrid.AbsoluteValue);

            public static explicit operator byte(Hybrid<T> hybrid) => Convert.ToByte(hybrid.Value);

            public static explicit operator sbyte(Hybrid<T> hybrid) =>
                Convert.ToSByte(hybrid.AbsoluteValue);

            public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
                default(T).ToString() : IsExternal ? $"<{AbsoluteValue}>" : Value.ToString();
        }
    }
```

./ILinks.cs

```csharp
using Platform.Data.Constants;

namespace Platform.Data.Doublets
{
    public interface ILinks<TLink> : ILinks<TLink, LinksCombinedConstants<TLink, TLink, int>>
    {
    }
}
```

./ILinksExtensions.cs

```csharp
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using Platform.Ranges;
using Platform.Collections.Arrays;
using Platform.Numbers;
using Platform.Random;
using Platform.Helpers.Setters;
using Platform.Data.Exceptions;

namespace Platform.Data.Doublets
{
    public static class ILinksExtensions
    {
        public static void RunRandomCreations<TLink>(this ILinks<TLink> links, long
            amountOfCreations)
        {
            for (long i = 0; i < amountOfCreations; i++)
            {
                var linksAddressRange = new Range<ulong>(0, (Integer<TLink>)links.Count());
                Integer<TLink> source = RandomHelpers.Default.NextUInt64(linksAddressRange);
                Integer<TLink> target = RandomHelpers.Default.NextUInt64(linksAddressRange);
                links.CreateAndUpdate(source, target);
            }
        }

        public static void RunRandomSearches<TLink>(this ILinks<TLink> links, long
            amountOfSearches)
        {
            for (long i = 0; i < amountOfSearches; i++)
            {
                var linkAddressRange = new Range<ulong>(1, (Integer<TLink>)links.Count());
                Integer<TLink> source = RandomHelpers.Default.NextUInt64(linkAddressRange);
                Integer<TLink> target = RandomHelpers.Default.NextUInt64(linkAddressRange);
                links.SearchOrDefault(source, target);
            }
        }

        public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, long
            amountOfDeletions)
        {
            var min = (ulong)amountOfDeletions > (Integer<TLink>)links.Count() ? 1 :
                (Integer<TLink>)links.Count() - (ulong)amountOfDeletions;
            for (long i = 0; i < amountOfDeletions; i++)
            {
                var linksAddressRange = new Range<ulong>(min, (Integer<TLink>)links.Count());
                Integer<TLink> link = RandomHelpers.Default.NextUInt64(linksAddressRange);
```

```csharp
                links.Delete(link);
                if ((Integer<TLink>)links.Count() < min)
                {
                    break;
                }
            }
        }

        /// <remarks>
        /// TODO: Возможно есть очень простой способ это сделать.
        /// (Например просто удалить файл, или изменить его размер таким образом,
        /// чтобы удалился весь контент)
        /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
        /// </remarks>
        public static void DeleteAll<TLink>(this ILinks<TLink> links)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            var comparer = Comparer<TLink>.Default;
            for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
            ↪   ArithmeticHelpers.Decrement(i))
            {
                links.Delete(i);
                if (!equalityComparer.Equals(links.Count(), ArithmeticHelpers.Decrement(i)))
                {
                    i = links.Count();
                }
            }
        }

        public static TLink First<TLink>(this ILinks<TLink> links)
        {
            TLink firstLink = default;
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (equalityComparer.Equals(links.Count(), default))
            {
                throw new Exception("В хранилище нет связей.");
            }
            links.Each(links.Constants.Any, links.Constants.Any, link =>
            {
                firstLink = link[links.Constants.IndexPart];
                return links.Constants.Break;
            });
            if (equalityComparer.Equals(firstLink, default))
            {
                throw new Exception("В процессе поиска по хранилищу не было найдено связей.");
            }
            return firstLink;
        }

        public static bool IsInnerReference<TLink>(this ILinks<TLink> links, TLink reference)
        {
            var constants = links.Constants;
            var comparer = Comparer<TLink>.Default;
            return comparer.Compare(constants.MinPossibleIndex, reference) >= 0 &&
            ↪   comparer.Compare(reference, constants.MaxPossibleIndex) <= 0;
        }

        #region Paths

        /// <remarks>
        /// TODO: Как так? Как то что ниже может быть корректно?
        /// Скорее всего практически не применимо
        /// Предполагалось, что можно было конвертировать формируемый в проходе через
        ↪   SequenceWalker
        /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
        /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
        /// </remarks>
        public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
        ↪   path)
        {
            var current = path[0];
            //EnsureLinkExists(current, "path");
            if (!links.Exists(current))
            {
                return false;
            }
            var equalityComparer = EqualityComparer<TLink>.Default;
            var constants = links.Constants;
            for (var i = 1; i < path.Length; i++)
```

```csharp
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
    ↪ SequenceWalker.
143         /// </remarks>
144         public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
    ↪ path)
145         {
146             links.EnsureLinkExists(root, "root");
147             var currentLink = root;
148             for (var i = 0; i < path.Length; i++)
149             {
150                 currentLink = links.GetLink(currentLink)[path[i]];
151             }
152             return currentLink;
153         }
154
155         public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
    ↪ links, TLink root, ulong size, ulong index)
156         {
157             var constants = links.Constants;
158             var source = constants.SourcePart;
159             var target = constants.TargetPart;
160             if (!MathHelpers.IsPowerOfTwo(size))
161             {
162                 throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
            ↪ than powers of two are not supported.");
163             }
164             var path = new BitArray(BitConverter.GetBytes(index));
165             var length = BitwiseHelpers.GetLowestBitPosition(size);
166             links.EnsureLinkExists(root, "root");
167             var currentLink = root;
168             for (var i = length - 1; i >= 0; i--)
169             {
170                 currentLink = links.GetLink(currentLink)[path[i] ? target : source];
171             }
172             return currentLink;
173         }
174
175         #endregion
176
177         /// <summary>
178         /// Возвращает индекс указанной связи.
179         /// </summary>
180         /// <param name="links">Хранилище связей.</param>
181         /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↪ содержимого.</param>
182         /// <returns>Индекс начальной связи для указанной связи.</returns>
183         [MethodImpl(MethodImplOptions.AggressiveInlining)]
184         public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↪ link[links.Constants.IndexPart];
185
186         /// <summary>
187         /// Возвращает индекс начальной (Source) связи для указанной связи.
188         /// </summary>
```

```csharp
            /// <param name="links">Хранилище связей.</param>
            /// <param name="link">Индекс связи.</param>
            /// <returns>Индекс начальной связи для указанной связи.</returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪    links.GetLink(link)[links.Constants.SourcePart];

            /// <summary>
            /// Возвращает индекс начальной (Source) связи для указанной связи.
            /// </summary>
            /// <param name="links">Хранилище связей.</param>
            /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪    содержимого.</param>
            /// <returns>Индекс начальной связи для указанной связи.</returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪    link[links.Constants.SourcePart];

            /// <summary>
            /// Возвращает индекс конечной (Target) связи для указанной связи.
            /// </summary>
            /// <param name="links">Хранилище связей.</param>
            /// <param name="link">Индекс связи.</param>
            /// <returns>Индекс конечной связи для указанной связи.</returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪    links.GetLink(link)[links.Constants.TargetPart];

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

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

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

```csharp
246            }

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

262
263            [MethodImpl(MethodImplOptions.AggressiveInlining)]
264            public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
             ↪  restrictions)
265            {
266                var constants = links.Constants;
267                int listSize = (Integer<TLink>)links.Count(restrictions);
268                var list = new IList<TLink>[listSize];
269                if (listSize > 0)
270                {
271                    var filler = new ArrayFiller<IList<TLink>, TLink>(list,
                     ↪  links.Constants.Continue);
272                    links.Each(filler.AddAndReturnConstant, restrictions);
273                }
274                return list;
275            }

276
277            /// <summary>
278            /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
             ↪  в хранилище связей.
279            /// </summary>
280            /// <param name="links">Хранилище связей.</param>
281            /// <param name="source">Начало связи.</param>
282            /// <param name="target">Конец связи.</param>
283            /// <returns>Значение, определяющее существует ли связь.</returns>
284            [MethodImpl(MethodImplOptions.AggressiveInlining)]
285            public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
             ↪  => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
             ↪  default) > 0;

286
287            #region Ensure
288            // TODO: May be move to EnsureExtensions or make it both there and here

289
290            [MethodImpl(MethodImplOptions.AggressiveInlining)]
291            public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
             ↪  reference, string argumentName)
292            {
293                if (links.IsInnerReference(reference) && !links.Exists(reference))
294                {
295                    throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
296                }
297            }

298
299            [MethodImpl(MethodImplOptions.AggressiveInlining)]
300            public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
             ↪  IList<TLink> restrictions, string argumentName)
301            {
302                for (int i = 0; i < restrictions.Count; i++)
303                {
304                    links.EnsureInnerReferenceExists(restrictions[i], argumentName);
305                }
306            }

307
308            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

```csharp
                }
                for (var i = 0; i < createdLinks.Count; i++)
                {
                    if (!nonExistentAddresses.Contains((Integer<TLink>)createdLinks[i]))
                    {
                        links.Delete(createdLinks[i]);
                    }
                }
            }
        }

        #endregion

        /// <param name="links">Хранилище связей.</param>
        public static ulong DependenciesCount<TLink>(this ILinks<TLink> links, TLink link)
        {
            var constants = links.Constants;
            var values = links.GetLink(link);
            ulong referencesAsSource = (Integer<TLink>)links.Count(constants.Any, link,
            ↪  constants.Any);
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (equalityComparer.Equals(values[constants.SourcePart], link))
            {
                referencesAsSource--;
            }
            ulong referencesAsTarget = (Integer<TLink>)links.Count(constants.Any, constants.Any,
            ↪  link);
            if (equalityComparer.Equals(values[constants.TargetPart], link))
            {
                referencesAsTarget--;
            }
            return referencesAsSource + referencesAsTarget;
        }

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool DependenciesExist<TLink>(this ILinks<TLink> links, TLink link) =>
        ↪  links.DependenciesCount(link) > 0;

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
        ↪  TLink target)
        {
            var constants = links.Constants;
            var values = links.GetLink(link);
            var equalityComparer = EqualityComparer<TLink>.Default;
            return equalityComparer.Equals(values[constants.SourcePart], source) &&
            ↪  equalityComparer.Equals(values[constants.TargetPart], target);
        }

        /// <summary>
        /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Индекс связи, которая является началом для искомой
        ↪  связи.</param>
        /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
        /// <returns>Индекс искомой связи с указанными Source (началом) и Target
        ↪  (концом).</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
        ↪  target)
        {
            var contants = links.Constants;
            var setter = new Setter<TLink, TLink>(contants.Continue, contants.Break, default);
            links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
            return setter.Result;
        }

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
        {
            var link = links.Create();
            return links.Update(link, link, link);
        }
```

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

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

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

./Incrementers/FrequencyIncrementer.cs

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

./Incrementers/LinkFrequencyIncrementer.cs

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

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

./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 =
   ↪   EqualityComparer<TLink>.Default;
 9
10            private readonly TLink _unaryOne;
11
12            public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
   ↪   _unaryOne = unaryOne;
13
14            public TLink Increment(TLink unaryNumber)
15            {
16                if (_equalityComparer.Equals(unaryNumber, _unaryOne))
17                {
18                    return Links.GetOrCreate(_unaryOne, _unaryOne);
19                }
20                var source = Links.GetSource(unaryNumber);
21                var target = Links.GetTarget(unaryNumber);
22                if (_equalityComparer.Equals(source, target))
23                {
24                    return Links.GetOrCreate(unaryNumber, _unaryOne);
25                }
26                else
27                {
28                    return Links.GetOrCreate(source, Increment(target));
29                }
30            }
31        }
32    }
```

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

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

```csharp
69          public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
    ↪   {source}->{target})";

70
71          public static string ToString(TLink source, TLink target) => $"({source}->{target})";

72
73          public static implicit operator TLink[](Link<TLink> link) => link.ToArray();

74
75          public static implicit operator Link<TLink>(TLink[] linkArray) => new
    ↪   Link<TLink>(linkArray);

76
77          public TLink[] ToArray()
78          {
79              var array = new TLink[Length];
80              CopyTo(array, 0);
81              return array;
82          }

83
84          public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
    ↪   ToString(Source, Target) : ToString(Index, Source, Target);

85
86          #region IList

87
88          public int Count => Length;

89
90          public bool IsReadOnly => true;

91
92          public TLink this[int index]
93          {
94              get
95              {
96                  Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
    ↪       nameof(index));
97                  if (index == _constants.IndexPart)
98                  {
99                      return Index;
100                 }
101                 if (index == _constants.SourcePart)
102                 {
103                     return Source;
104                 }
105                 if (index == _constants.TargetPart)
106                 {
107                     return Target;
108                 }
109                 throw new NotSupportedException(); // Impossible path due to
    ↪       Ensure.ArgumentInRange
110             }
111             set => throw new NotSupportedException();
112         }

113
114         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

115
116         public IEnumerator<TLink> GetEnumerator()
117         {
118             yield return Index;
119             yield return Source;
120             yield return Target;
121         }

122
123         public void Add(TLink item) => throw new NotSupportedException();

124
125         public void Clear() => throw new NotSupportedException();

126
127         public bool Contains(TLink item) => IndexOf(item) >= 0;

128
129         public void CopyTo(TLink[] array, int arrayIndex)
130         {
131             Ensure.Always.ArgumentNotNull(array, nameof(array));
132             Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
    ↪       nameof(arrayIndex));
133             if (arrayIndex + Length > array.Length)
134             {
135                 throw new InvalidOperationException();
136             }
137             array[arrayIndex++] = Index;
138             array[arrayIndex++] = Source;
139             array[arrayIndex] = Target;
140         }

141
```

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

./LinkExtensions.cs

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

./LinksOperatorBase.cs

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

./PropertyOperators/DefaultLinkPropertyOperator.cs

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

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

./PropertyOperators/FrequencyPropertyOperator.cs

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

```
62              }
63          }
64  }
```

./ResizableDirectMemory/ResizableDirectMemoryLinks.cs

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

```csharp
                    public static TLink GetLeftAsSource(IntPtr pointer) => (pointer +
                    ↪ LeftAsSourceOffset).GetValue<TLink>();
                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    public static TLink GetRightAsSource(IntPtr pointer) => (pointer +
                    ↪ RightAsSourceOffset).GetValue<TLink>();
                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    public static TLink GetSizeAsSource(IntPtr pointer) => (pointer +
                    ↪ SizeAsSourceOffset).GetValue<TLink>();
                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    public static TLink GetLeftAsTarget(IntPtr pointer) => (pointer +
                    ↪ LeftAsTargetOffset).GetValue<TLink>();
                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    public static TLink GetRightAsTarget(IntPtr pointer) => (pointer +
                    ↪ RightAsTargetOffset).GetValue<TLink>();
                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    public static TLink GetSizeAsTarget(IntPtr pointer) => (pointer +
                    ↪ SizeAsTargetOffset).GetValue<TLink>();

                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    public static void SetSource(IntPtr pointer, TLink value) => (pointer +
                    ↪ SourceOffset).SetValue(value);
                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    public static void SetTarget(IntPtr pointer, TLink value) => (pointer +
                    ↪ TargetOffset).SetValue(value);
                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    public static void SetLeftAsSource(IntPtr pointer, TLink value) => (pointer +
                    ↪ LeftAsSourceOffset).SetValue(value);
                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    public static void SetRightAsSource(IntPtr pointer, TLink value) => (pointer +
                    ↪ RightAsSourceOffset).SetValue(value);
                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    public static void SetSizeAsSource(IntPtr pointer, TLink value) => (pointer +
                    ↪ SizeAsSourceOffset).SetValue(value);
                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    public static void SetLeftAsTarget(IntPtr pointer, TLink value) => (pointer +
                    ↪ LeftAsTargetOffset).SetValue(value);
                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    public static void SetRightAsTarget(IntPtr pointer, TLink value) => (pointer +
                    ↪ RightAsTargetOffset).SetValue(value);
                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    public static void SetSizeAsTarget(IntPtr pointer, TLink value) => (pointer +
                    ↪ SizeAsTargetOffset).SetValue(value);
                }

        private struct LinksHeader
        {
            public static readonly int AllocatedLinksOffset =
            ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(AllocatedLinks)).ToInt32();
            public static readonly int ReservedLinksOffset =
            ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(ReservedLinks)).ToInt32();
            public static readonly int FreeLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
            ↪ nameof(FreeLinks)).ToInt32();
            public static readonly int FirstFreeLinkOffset =
            ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstFreeLink)).ToInt32();
            public static readonly int FirstAsSourceOffset =
            ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsSource)).ToInt32();
            public static readonly int FirstAsTargetOffset =
            ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsTarget)).ToInt32();
            public static readonly int LastFreeLinkOffset =
            ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(LastFreeLink)).ToInt32();

            public TLink AllocatedLinks;
            public TLink ReservedLinks;
            public TLink FreeLinks;
            public TLink FirstFreeLink;
            public TLink FirstAsSource;
            public TLink FirstAsTarget;
            public TLink LastFreeLink;
            public TLink Reserved8;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetAllocatedLinks(IntPtr pointer) => (pointer +
            ↪ AllocatedLinksOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetReservedLinks(IntPtr pointer) => (pointer +
            ↪ ReservedLinksOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
            public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
            ↪  FreeLinksOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
            ↪  FirstFreeLinkOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
            ↪  FirstAsSourceOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
            ↪  FirstAsTargetOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetLastFreeLink(IntPtr pointer) => (pointer +
            ↪  LastFreeLinkOffset).GetValue<TLink>();

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) => pointer +
            ↪  FirstAsSourceOffset;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
            ↪  FirstAsTargetOffset;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
            ↪  AllocatedLinksOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
            ↪  ReservedLinksOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
            ↪  FreeLinksOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
            ↪  FirstFreeLinkOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
            ↪  FirstAsSourceOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
            ↪  FirstAsTargetOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
            ↪  LastFreeLinkOffset).SetValue(value);
        }

        private readonly long _memoryReservationStep;

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

        private LinksTargetsTreeMethods _targetsTreeMethods;
        private LinksSourcesTreeMethods _sourcesTreeMethods;

        // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
        ↪  нужно использовать не список а дерево, так как так можно быстрее проверить на
        ↪  наличие связи внутри
        private UnusedLinksListMethods _unusedLinksListMethods;

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

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

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

        /// <summary>
        /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
        ↪  минимальным шагом расширения базы данных.
        /// </summary>
        /// <param name="address">Полный пусть к файлу базы данных.</param>
```

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

```csharp
321             if (restrictions.Count == 0)
322             {
323                 for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
    ↪   (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
    ↪   Increment(link))
324                 {
325                     if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
    ↪   Constants.Break))
326                     {
327                         return Constants.Break;
328                     }
329                 }
330
331                 return Constants.Continue;
332             }
333             if (restrictions.Count == 1)
334             {
335                 var index = restrictions[Constants.IndexPart];
336                 if (_equalityComparer.Equals(index, Constants.Any))
337                 {
338                     return Each(handler, ArrayPool<TLink>.Empty);
339                 }
340                 if (!Exists(index))
341                 {
342                     return Constants.Continue;
343                 }
344                 return handler(GetLinkStruct(index));
345             }
346             if (restrictions.Count == 2)
347             {
348                 var index = restrictions[Constants.IndexPart];
349                 var value = restrictions[1];
350                 if (_equalityComparer.Equals(index, Constants.Any))
351                 {
352                     if (_equalityComparer.Equals(value, Constants.Any))
353                     {
354                         return Each(handler, ArrayPool<TLink>.Empty);
355                     }
356                     if (_equalityComparer.Equals(Each(handler, new[] { index, value,
    ↪   Constants.Any }), Constants.Break))
357                     {
358                         return Constants.Break;
359                     }
360                     return Each(handler, new[] { index, Constants.Any, value });
361                 }
362                 else
363                 {
364                     if (!Exists(index))
365                     {
366                         return Constants.Continue;
367                     }
368                     if (_equalityComparer.Equals(value, Constants.Any))
369                     {
370                         return handler(GetLinkStruct(index));
371                     }
372                     var storedLinkValue = GetLinkUnsafe(index);
373                     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
374                         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
375                     {
376                         return handler(GetLinkStruct(index));
377                     }
378                     return Constants.Continue;
379                 }
380             }
381             if (restrictions.Count == 3)
382             {
383                 var index = restrictions[Constants.IndexPart];
384                 var source = restrictions[Constants.SourcePart];
385                 var target = restrictions[Constants.TargetPart];
386                 if (_equalityComparer.Equals(index, Constants.Any))
387                 {
388                     if (_equalityComparer.Equals(source, Constants.Any) &&
    ↪   _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.EachReference(target, handler);
395                        }
396                    else if (_equalityComparer.Equals(target, Constants.Any))
397                        {
398                            return _sourcesTreeMethods.EachReference(source, handler);
399                        }
400                    else //if(source != Any && target != Any)
401                        {
402                            var link = _sourcesTreeMethods.Search(source, target);
403                            return _equalityComparer.Equals(link, Constants.Null) ?
       ↪                Constants.Continue : handler(GetLinkStruct(link));
404                        }
405                    }
406                else
407                    {
408                        if (!Exists(index))
409                        {
410                            return Constants.Continue;
411                        }
412                        if (_equalityComparer.Equals(source, Constants.Any) &&
       ↪                _equalityComparer.Equals(target, Constants.Any))
413                        {
414                            return handler(GetLinkStruct(index));
415                        }
416                        var storedLinkValue = GetLinkUnsafe(index);
417                        if (!_equalityComparer.Equals(source, Constants.Any) &&
       ↪                !_equalityComparer.Equals(target, Constants.Any))
418                        {
419                            if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
420                                _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
421                            {
422                                return handler(GetLinkStruct(index));
423                            }
424                            return Constants.Continue;
425                        }
426                        var value = default(TLink);
427                        if (_equalityComparer.Equals(source, Constants.Any))
428                        {
429                            value = target;
430                        }
431                        if (_equalityComparer.Equals(target, Constants.Any))
432                        {
433                            value = source;
434                        }
435                        if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
436                            _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
437                        {
438                            return handler(GetLinkStruct(index));
439                        }
440                        return Constants.Continue;
441                    }
442                }
443            throw new NotSupportedException("Другие размеры и способы ограничений не
       ↪        поддерживаются.");
444        }
445
446        /// <remarks>
447        /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
       ↪    в другом месте (но не в менеджере памяти, а в логике Links)
448        /// </remarks>
449        [MethodImpl(MethodImplOptions.AggressiveInlining)]
450        public TLink Update(IList<TLink> values)
451        {
452            var linkIndex = values[Constants.IndexPart];
453            var link = GetLinkUnsafe(linkIndex);
454            // Будет корректно работать только в том случае, если пространство выделенной связи
       ↪        предварительно заполнено нулями
455            if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
456            {
457                _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
       ↪            linkIndex);
458            }
459            if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
460            {
461                _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
       ↪            linkIndex);
462            }
463            Link.SetSource(link, values[Constants.SourcePart]);
```

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

```csharp
                        _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
                        LinksHeader.SetAllocatedLinks(_header,
                        ↪   Decrement(LinksHeader.GetAllocatedLinks(_header)));
                        _memory.UsedCapacity -= LinkSizeInBytes;
                    }
                }
            }

            /// <remarks>
            /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
            ↪   адрес реально поменялся
            ///
            /// Указатель this.links может быть в том же месте,
            /// так как 0-я связь не используется и имеет такой же размер как Header,
            /// поэтому header размещается в том же месте, что и 0-я связь
            /// </remarks>
            private void SetPointers(IDirectMemory memory)
            {
                if (memory == null)
                {
                    _links = IntPtr.Zero;
                    _header = _links;
                    _unusedLinksListMethods = null;
                    _targetsTreeMethods = null;
                    _unusedLinksListMethods = null;
                }
                else
                {
                    _links = memory.Pointer;
                    _header = _links;
                    _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
                    _targetsTreeMethods = new LinksTargetsTreeMethods(this);
                    _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool Exists(TLink link)
                => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
                && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
                && !IsUnusedLink(link);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool IsUnusedLink(TLink link)
                => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
                || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)),
                ↪   Constants.Null)
                && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)), Constants.Null));

            #region DisposableBase

            protected override bool AllowMultipleDisposeCalls => true;

            protected override void DisposeCore(bool manual, bool wasDisposed)
            {
                if (!wasDisposed)
                {
                    SetPointers(null);
                }
                Disposable.TryDispose(_memory);
            }

            #endregion
        }
    }
```

./ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs

```csharp
using System;
using Platform.Unsafe;
using Platform.Collections.Methods.Lists;

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    partial class ResizableDirectMemoryLinks<TLink>
    {
        private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<TLink>
        {
            private readonly IntPtr _links;
            private readonly IntPtr _header;
```

```csharp
                    public UnusedLinksListMethods(IntPtr links, IntPtr header)
                    {
                        _links = links;
                        _header = header;
                    }

                    protected override TLink GetFirst() => (_header +
                    ↪  LinksHeader.FirstFreeLinkOffset).GetValue<TLink>();

                    protected override TLink GetLast() => (_header +
                    ↪  LinksHeader.LastFreeLinkOffset).GetValue<TLink>();

                    protected override TLink GetPrevious(TLink element) =>
                    ↪  (_links.GetElement(LinkSizeInBytes, element) +
                    ↪  Link.SourceOffset).GetValue<TLink>();

                    protected override TLink GetNext(TLink element) =>
                    ↪  (_links.GetElement(LinkSizeInBytes, element) +
                    ↪  Link.TargetOffset).GetValue<TLink>();

                    protected override TLink GetSize() => (_header +
                    ↪  LinksHeader.FreeLinksOffset).GetValue<TLink>();

                    protected override void SetFirst(TLink element) => (_header +
                    ↪  LinksHeader.FirstFreeLinkOffset).SetValue(element);

                    protected override void SetLast(TLink element) => (_header +
                    ↪  LinksHeader.LastFreeLinkOffset).SetValue(element);

                    protected override void SetPrevious(TLink element, TLink previous) =>
                    ↪  (_links.GetElement(LinkSizeInBytes, element) +
                    ↪  Link.SourceOffset).SetValue(previous);

                    protected override void SetNext(TLink element, TLink next) =>
                    ↪  (_links.GetElement(LinkSizeInBytes, element) + Link.TargetOffset).SetValue(next);

                    protected override void SetSize(TLink size) => (_header +
                    ↪  LinksHeader.FreeLinksOffset).SetValue(size);
                }
            }
        }
```

./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs

```csharp
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Numbers;
using Platform.Unsafe;
using Platform.Collections.Methods.Trees;
using Platform.Data.Constants;

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    partial class ResizableDirectMemoryLinks<TLink>
    {
        private abstract class LinksTreeMethodsBase :
        ↪  SizedAndThreadedAVLBalancedTreeMethods<TLink>
        {
            private readonly ResizableDirectMemoryLinks<TLink> _memory;
            private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
            protected readonly IntPtr Links;
            protected readonly IntPtr Header;

            protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
            {
                Links = memory._links;
                Header = memory._header;
                _memory = memory;
                _constants = memory.Constants;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract TLink GetTreeRoot();

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract TLink GetBasePartValue(TLink link);

            public TLink this[TLink index]
            {
```

```csharp
                    get
                    {
                        var root = GetTreeRoot();
                        if (GreaterOrEqualThan(index, GetSize(root)))
                        {
                            return GetZero();
                        }
                        while (!EqualToZero(root))
                        {
                            var left = GetLeftOrDefault(root);
                            var leftSize = GetSizeOrZero(left);
                            if (LessThan(index, leftSize))
                            {
                                root = left;
                                continue;
                            }
                            if (IsEquals(index, leftSize))
                            {
                                return root;
                            }
                            root = GetRightOrDefault(root);
                            index = Subtract(index, Increment(leftSize));
                        }
                        return GetZero(); // TODO: Impossible situation exception (only if tree
                        ↪  structure broken)
                    }
                }

        // TODO: Return indices range instead of references count
        public TLink CalculateReferences(TLink link)
        {
            var root = GetTreeRoot();
            var total = GetSize(root);
            var totalRightIgnore = GetZero();
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (LessOrEqualThan(@base, link))
                {
                    root = GetRightOrDefault(root);
                }
                else
                {
                    totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
                    root = GetLeftOrDefault(root);
                }
            }
            root = GetTreeRoot();
            var totalLeftIgnore = GetZero();
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (GreaterOrEqualThan(@base, link))
                {
                    root = GetLeftOrDefault(root);
                }
                else
                {
                    totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));

                    root = GetRightOrDefault(root);
                }
            }
            return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
        }

        public TLink EachReference(TLink link, Func<IList<TLink>, TLink> handler)
        {
            var root = GetTreeRoot();
            if (EqualToZero(root))
            {
                return _constants.Continue;
            }
            TLink first = GetZero(), current = root;
            while (!EqualToZero(current))
            {
                var @base = GetBasePartValue(current);
                if (GreaterOrEqualThan(@base, link))
```

```csharp
                    {
                        if (IsEquals(@base, link))
                        {
                            first = current;
                        }
                        current = GetLeftOrDefault(current);
                    }
                    else
                    {
                        current = GetRightOrDefault(current);
                    }
                }
                if (!EqualToZero(first))
                {
                    current = first;
                    while (true)
                    {
                        if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
                        {
                            return _constants.Break;
                        }
                        current = GetNext(current);
                        if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
                        {
                            break;
                        }
                    }
                }
                return _constants.Continue;
            }

            protected override void PrintNodeValue(TLink node, StringBuilder sb)
            {
                sb.Append(' ');
                sb.Append((Links.GetElement(LinkSizeInBytes, node) +
                →  Link.SourceOffset).GetValue<TLink>());
                sb.Append('-');
                sb.Append('>');
                sb.Append((Links.GetElement(LinkSizeInBytes, node) +
                →  Link.TargetOffset).GetValue<TLink>());
            }
        }

        private class LinksSourcesTreeMethods : LinksTreeMethodsBase
        {
            public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
                : base(memory)
            {
            }

            protected override IntPtr GetLeftPointer(TLink node) =>
            →  Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset;

            protected override IntPtr GetRightPointer(TLink node) =>
            →  Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset;

            protected override TLink GetLeftValue(TLink node) =>
            →  (Links.GetElement(LinkSizeInBytes, node) +
            →  Link.LeftAsSourceOffset).GetValue<TLink>();

            protected override TLink GetRightValue(TLink node) =>
            →  (Links.GetElement(LinkSizeInBytes, node) +
            →  Link.RightAsSourceOffset).GetValue<TLink>();

            protected override TLink GetSize(TLink node)
            {
                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                →  Link.SizeAsSourceOffset).GetValue<TLink>();
                return BitwiseHelpers.PartialRead(previousValue, 5, -5);
            }

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

            protected override void SetRight(TLink node, TLink right) =>
            →  (Links.GetElement(LinkSizeInBytes, node) +
            →  Link.RightAsSourceOffset).SetValue(right);
```

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

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

```csharp
287          public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
288              : base(memory)
289          {
290          }
291
292          protected override IntPtr GetLeftPointer(TLink node) =>
    ↪      Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset;
293
294          protected override IntPtr GetRightPointer(TLink node) =>
    ↪      Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset;
295
296          protected override TLink GetLeftValue(TLink node) =>
    ↪      (Links.GetElement(LinkSizeInBytes, node) +
    ↪      Link.LeftAsTargetOffset).GetValue<TLink>();
297
298          protected override TLink GetRightValue(TLink node) =>
    ↪      (Links.GetElement(LinkSizeInBytes, node) +
    ↪      Link.RightAsTargetOffset).GetValue<TLink>();
299
300          protected override TLink GetSize(TLink node)
301          {
302              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
    ↪          Link.SizeAsTargetOffset).GetValue<TLink>();
303              return BitwiseHelpers.PartialRead(previousValue, 5, -5);
304          }
305
306          protected override void SetLeft(TLink node, TLink left) =>
    ↪      (Links.GetElement(LinkSizeInBytes, node) +
    ↪      Link.LeftAsTargetOffset).SetValue(left);
307
308          protected override void SetRight(TLink node, TLink right) =>
    ↪      (Links.GetElement(LinkSizeInBytes, node) +
    ↪      Link.RightAsTargetOffset).SetValue(right);
309
310          protected override void SetSize(TLink node, TLink size)
311          {
312              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
    ↪          Link.SizeAsTargetOffset).GetValue<TLink>();
313              (Links.GetElement(LinkSizeInBytes, node) +
    ↪          Link.SizeAsTargetOffset).SetValue(BitwiseHelpers.PartialWrite(previousValue,
    ↪          size, 5, -5));
314          }
315
316          protected override bool GetLeftIsChild(TLink node)
317          {
318              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
    ↪          Link.SizeAsTargetOffset).GetValue<TLink>();
319              return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 4, 1);
320          }
321
322          protected override void SetLeftIsChild(TLink node, bool value)
323          {
324              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
    ↪          Link.SizeAsTargetOffset).GetValue<TLink>();
325              var modified = BitwiseHelpers.PartialWrite(previousValue,
    ↪          (TLink)(Integer<TLink>)value, 4, 1);
326              (Links.GetElement(LinkSizeInBytes, node) +
    ↪          Link.SizeAsTargetOffset).SetValue(modified);
327          }
328
329          protected override bool GetRightIsChild(TLink node)
330          {
331              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
    ↪          Link.SizeAsTargetOffset).GetValue<TLink>();
332              return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 3, 1);
333          }
334
335          protected override void SetRightIsChild(TLink node, bool value)
336          {
337              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
    ↪          Link.SizeAsTargetOffset).GetValue<TLink>();
338              var modified = BitwiseHelpers.PartialWrite(previousValue,
    ↪          (TLink)(Integer<TLink>)value, 3, 1);
339              (Links.GetElement(LinkSizeInBytes, node) +
    ↪          Link.SizeAsTargetOffset).SetValue(modified);
340          }
341
```

```csharp
                protected override sbyte GetBalance(TLink node)
                {
                    var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsTargetOffset).GetValue<TLink>();
                    var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 0,
                    ↪  3);
                    var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                    ↪  124 : value & 3);
                    return unpackedValue;
                }

                protected override void SetBalance(TLink node, sbyte value)
                {
                    var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsTargetOffset).GetValue<TLink>();
                    var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
                    ↪  3);
                    var modified = BitwiseHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
                    (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsTargetOffset).SetValue(modified);
                }

                protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
                {
                    var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
                    ↪  Link.TargetOffset).GetValue<TLink>();
                    var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
                    ↪  Link.TargetOffset).GetValue<TLink>();
                    return LessThan(firstTarget, secondTarget) ||
                        (IsEquals(firstTarget, secondTarget) &&
                            LessThan((Links.GetElement(LinkSizeInBytes, first) +
                    ↪  Link.SourceOffset).GetValue<TLink>(),
                    ↪  (Links.GetElement(LinkSizeInBytes, second) +
                    ↪  Link.SourceOffset).GetValue<TLink>()));
                }

                protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
                {
                    var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
                    ↪  Link.TargetOffset).GetValue<TLink>();
                    var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
                    ↪  Link.TargetOffset).GetValue<TLink>();
                    return GreaterThan(firstTarget, secondTarget) ||
                        (IsEquals(firstTarget, secondTarget) &&
                            GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
                    ↪  Link.SourceOffset).GetValue<TLink>(),
                    ↪  (Links.GetElement(LinkSizeInBytes, second) +
                    ↪  Link.SourceOffset).GetValue<TLink>()));
                }

                protected override TLink GetTreeRoot() => (Header +
                ↪  LinksHeader.FirstAsTargetOffset).GetValue<TLink>();

                protected override TLink GetBasePartValue(TLink link) =>
                ↪  (Links.GetElement(LinkSizeInBytes, link) + Link.TargetOffset).GetValue<TLink>();
            }
        }
    }
```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs

```csharp
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Disposables;
using Platform.Collections.Arrays;
using Platform.Helpers.Singletons;
using Platform.Memory;
using Platform.Data.Exceptions;
using Platform.Data.Constants;

//#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION

#pragma warning disable 0649
#pragma warning disable 169

// ReSharper disable BuiltInTypeReferenceStyle

namespace Platform.Data.Doublets.ResizableDirectMemory
```

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

```csharp
            _memory = memory;
            _memoryReservationStep = memoryReservationStep;
            if (memory.ReservedCapacity < memoryReservationStep)
            {
                memory.ReservedCapacity = memoryReservationStep;
            }
            SetPointers(_memory);
            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
            _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) +
            ↪  sizeof(LinksHeader);
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(LinksHeader)) /
            ↪  sizeof(Link));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public id Count(IList<id> restrictions)
        {
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
            if (restrictions.Count == 0)
            {
                return Total;
            }
            if (restrictions.Count == 1)
            {
                var index = restrictions[Constants.IndexPart];
                if (index == Constants.Any)
                {
                    return Total;
                }
                return Exists(index) ? 1UL : 0UL;
            }
            if (restrictions.Count == 2)
            {
                var index = restrictions[Constants.IndexPart];
                var value = restrictions[1];
                if (index == Constants.Any)
                {
                    if (value == Constants.Any)
                    {
                        return Total; // Any - как отсутствие ограничения
                    }
                    return _sourcesTreeMethods.CalculateReferences(value)
                        + _targetsTreeMethods.CalculateReferences(value);
                }
                else
                {
                    if (!Exists(index))
                    {
                        return 0;
                    }
                    if (value == Constants.Any)
                    {
                        return 1;
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (storedLinkValue->Source == value ||
                        storedLinkValue->Target == value)
                    {
                        return 1;
                    }
                    return 0;
                }
            }
            if (restrictions.Count == 3)
            {
                var index = restrictions[Constants.IndexPart];
                var source = restrictions[Constants.SourcePart];
                var target = restrictions[Constants.TargetPart];
                if (index == Constants.Any)
                {
                    if (source == Constants.Any && target == Constants.Any)
                    {
                        return Total;
                    }
                    else if (source == Constants.Any)
                    {
                        return _targetsTreeMethods.CalculateReferences(target);
```

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

```csharp
                            return Constants.Continue;
                        }
                        return handler(GetLinkStruct(index));
                    }
                    if (restrictions.Count == 2)
                    {
                        var index = restrictions[Constants.IndexPart];
                        var value = restrictions[1];
                        if (index == Constants.Any)
                        {
                            if (value == Constants.Any)
                            {
                                return Each(handler, ArrayPool<ulong>.Empty);
                            }
                            if (Each(handler, new[] { index, value, Constants.Any }) == Constants.Break)
                            {
                                return Constants.Break;
                            }
                            return Each(handler, new[] { index, Constants.Any, value });
                        }
                        else
                        {
                            if (!Exists(index))
                            {
                                return Constants.Continue;
                            }
                            if (value == Constants.Any)
                            {
                                return handler(GetLinkStruct(index));
                            }
                            var storedLinkValue = GetLinkUnsafe(index);
                            if (storedLinkValue->Source == value ||
                                storedLinkValue->Target == value)
                            {
                                return handler(GetLinkStruct(index));
                            }
                            return Constants.Continue;
                        }
                    }
                    if (restrictions.Count == 3)
                    {
                        var index = restrictions[Constants.IndexPart];
                        var source = restrictions[Constants.SourcePart];
                        var target = restrictions[Constants.TargetPart];
                        if (index == Constants.Any)
                        {
                            if (source == Constants.Any && target == Constants.Any)
                            {
                                return Each(handler, ArrayPool<ulong>.Empty);
                            }
                            else if (source == Constants.Any)
                            {
                                return _targetsTreeMethods.EachReference(target, handler);
                            }
                            else if (target == Constants.Any)
                            {
                                return _sourcesTreeMethods.EachReference(source, handler);
                            }
                            else //if(source != Any && target != Any)
                            {
                                var link = _sourcesTreeMethods.Search(source, target);
                                return link == Constants.Null ? Constants.Continue :
                                ↪  handler(GetLinkStruct(link));
                            }
                        }
                        else
                        {
                            if (!Exists(index))
                            {
                                return Constants.Continue;
                            }
                            if (source == Constants.Any && target == Constants.Any)
                            {
                                return handler(GetLinkStruct(index));
                            }
                            var storedLinkValue = GetLinkUnsafe(index);
                            if (source != Constants.Any && target != Constants.Any)
                            {
```

```csharp
                            if (storedLinkValue->Source == source &&
                                storedLinkValue->Target == target)
                            {
                                return handler(GetLinkStruct(index));
                            }
                            return Constants.Continue;
                        }
                        var value = default(id);
                        if (source == Constants.Any)
                        {
                            value = target;
                        }
                        if (target == Constants.Any)
                        {
                            value = source;
                        }
                        if (storedLinkValue->Source == value ||
                            storedLinkValue->Target == value)
                        {
                            return handler(GetLinkStruct(index));
                        }
                        return Constants.Continue;
                    }
                }
                throw new NotSupportedException("Другие размеры и способы ограничений не
                    поддерживаются.");
            }

            /// <remarks>
            /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
            ///   в другом месте (но не в менеджере памяти, а в логике Links)
            /// </remarks>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public id Update(IList<id> values)
            {
                var linkIndex = values[Constants.IndexPart];
                var link = GetLinkUnsafe(linkIndex);
                // Будет корректно работать только в том случае, если пространство выделенной связи
                //   предварительно заполнено нулями
                if (link->Source != Constants.Null)
                {
                    _sourcesTreeMethods.Detach(new IntPtr(&_header->FirstAsSource), linkIndex);
                }
                if (link->Target != Constants.Null)
                {
                    _targetsTreeMethods.Detach(new IntPtr(&_header->FirstAsTarget), linkIndex);
                }
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
                var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
                var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
                if (leftTreeSize != rightTreeSize)
                {
                    throw new Exception("One of the trees is broken.");
                }
#endif
                link->Source = values[Constants.SourcePart];
                link->Target = values[Constants.TargetPart];
                if (link->Source != Constants.Null)
                {
                    _sourcesTreeMethods.Attach(new IntPtr(&_header->FirstAsSource), linkIndex);
                }
                if (link->Target != Constants.Null)
                {
                    _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
                }
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
                leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
                rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
                if (leftTreeSize != rightTreeSize)
                {
                    throw new Exception("One of the trees is broken.");
                }
#endif
                return linkIndex;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private IList<id> GetLinkStruct(id linkIndex)
```

```csharp
397         {
398             var link = GetLinkUnsafe(linkIndex);
399             return new UInt64Link(linkIndex, link->Source, link->Target);
400         }
401
402         [MethodImpl(MethodImplOptions.AggressiveInlining)]
403         private Link* GetLinkUnsafe(id linkIndex) => &_links[linkIndex];
404
405         /// <remarks>
406         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
        ↪ пространство
407         /// </remarks>
408         public id Create()
409         {
410             var freeLink = _header->FirstFreeLink;
411             if (freeLink != Constants.Null)
412             {
413                 _unusedLinksListMethods.Detach(freeLink);
414             }
415             else
416             {
417                 if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
418                 {
419                     throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
420                 }
421                 if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
422                 {
423                     _memory.ReservedCapacity += _memoryReservationStep;
424                     SetPointers(_memory);
425                     _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
426                 }
427                 _header->AllocatedLinks++;
428                 _memory.UsedCapacity += sizeof(Link);
429                 freeLink = _header->AllocatedLinks;
430             }
431             return freeLink;
432         }
433
434         public void Delete(id link)
435         {
436             if (link < _header->AllocatedLinks)
437             {
438                 _unusedLinksListMethods.AttachAsFirst(link);
439             }
440             else if (link == _header->AllocatedLinks)
441             {
442                 _header->AllocatedLinks--;
443                 _memory.UsedCapacity -= sizeof(Link);
444                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                ↪ пока не дойдём до первой существующей связи
445                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
446                 while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
447                 {
448                     _unusedLinksListMethods.Detach(_header->AllocatedLinks);
449                     _header->AllocatedLinks--;
450                     _memory.UsedCapacity -= sizeof(Link);
451                 }
452             }
453         }
454
455         /// <remarks>
456         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
        ↪ адрес реально поменялся
457         ///
458         /// Указатель this.links может быть в том же месте,
459         /// так как 0-я связь не используется и имеет такой же размер как Header,
460         /// поэтому header размещается в том же месте, что и 0-я связь
461         /// </remarks>
462         private void SetPointers(IResizableDirectMemory memory)
463         {
464             if (memory == null)
465             {
466                 _header = null;
467                 _links = null;
468                 _unusedLinksListMethods = null;
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
487                                                   || (_links[link].SizeAsSource == Constants.Null &&
    ↪   _links[link].Source != Constants.Null);
488
489                #region Disposable
490
491                protected override bool AllowMultipleDisposeCalls => true;
492
493                protected override void DisposeCore(bool manual, bool wasDisposed)
494                {
495                    if (!wasDisposed)
496                    {
497                        SetPointers(null);
498                    }
499                    Disposable.TryDispose(_memory);
500                }
501
502                #endregion
503            }
504    }
```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs

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

```csharp
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Text;
using Platform.Collections.Methods.Trees;
using Platform.Data.Constants;

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    unsafe partial class UInt64ResizableDirectMemoryLinks
    {
        private abstract class LinksTreeMethodsBase :
            SizedAndThreadedAVLBalancedTreeMethods<ulong>
        {
            private readonly UInt64ResizableDirectMemoryLinks _memory;
            private readonly LinksCombinedConstants<ulong, ulong, int> _constants;
            protected readonly Link* Links;
            protected readonly LinksHeader* Header;

            protected LinksTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory)
            {
                Links = memory._links;
                Header = memory._header;
                _memory = memory;
                _constants = memory.Constants;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract ulong GetTreeRoot();

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract ulong GetBasePartValue(ulong link);

            public ulong this[ulong index]
            {
                get
                {
                    var root = GetTreeRoot();
                    if (index >= GetSize(root))
                    {
                        return 0;
                    }
                    while (root != 0)
                    {
                        var left = GetLeftOrDefault(root);
                        var leftSize = GetSizeOrZero(left);
                        if (index < leftSize)
                        {
                            root = left;
                            continue;
                        }
                        if (index == leftSize)
                        {
                            return root;
                        }
                        root = GetRightOrDefault(root);
                        index -= leftSize + 1;
                    }
                    return 0; // TODO: Impossible situation exception (only if tree structure
                              // broken)
                }
            }

            // TODO: Return indices range instead of references count
            public ulong CalculateReferences(ulong link)
            {
                var root = GetTreeRoot();
                var total = GetSize(root);
                var totalRightIgnore = 0UL;
                while (root != 0)
                {
                    var @base = GetBasePartValue(root);
                    if (@base <= link)
                    {
                        root = GetRightOrDefault(root);
                    }
                    else
                    {
                        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        private class LinksSourcesTreeMethods : LinksTreeMethodsBase
153        {
154            public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)
155                : base(memory)
156            {
```

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

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

```csharp
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override ulong GetOne() => 1UL;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override ulong GetTwo() => 2UL;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool ValueEqualToZero(IntPtr pointer) =>
            ↪  *(ulong*)pointer.ToPointer() == 0UL;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool EqualToZero(ulong value) => value == 0UL;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool IsEquals(ulong first, ulong second) => first == second;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool GreaterThanZero(ulong value) => value > 0UL;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool GreaterThan(ulong first, ulong second) => first > second;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >=
            ↪  second;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0
            ↪  is always true for ulong

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is
            ↪  always >= 0 for ulong

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool LessOrEqualThan(ulong first, ulong second) => first <=
            ↪  second;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool LessThanZero(ulong value) => false; // value < 0 is always
            ↪  false for ulong

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool LessThan(ulong first, ulong second) => first < second;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override ulong Increment(ulong value) => ++value;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override ulong Decrement(ulong value) => --value;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override ulong Add(ulong first, ulong second) => first + second;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override ulong Subtract(ulong first, ulong second) => first - second;
        }

        private class LinksTargetsTreeMethods : LinksTreeMethodsBase
        {
            public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
                : base(memory)
            {
            }

            //protected override IntPtr GetLeft(ulong node) => new
            ↪  IntPtr(&Links[node].LeftAsTarget);

            //protected override IntPtr GetRight(ulong node) => new
            ↪  IntPtr(&Links[node].RightAsTarget);

            //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;

            //protected override void SetLeft(ulong node, ulong left) =>
            ↪  Links[node].LeftAsTarget = left;

            //protected override void SetRight(ulong node, ulong right) =>
            ↪  Links[node].RightAsTarget = right;
```

```csharp
        //protected override void SetSize(ulong node, ulong size) =>
        ↪  Links[node].SizeAsTarget = size;

        protected override IntPtr GetLeftPointer(ulong node) => new
        ↪  IntPtr(&Links[node].LeftAsTarget);

        protected override IntPtr GetRightPointer(ulong node) => new
        ↪  IntPtr(&Links[node].RightAsTarget);

        protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsTarget;

        protected override ulong GetRightValue(ulong node) => Links[node].RightAsTarget;

        protected override ulong GetSize(ulong node)
        {
            var previousValue = Links[node].SizeAsTarget;
            //return MathHelpers.PartialRead(previousValue, 5, -5);
            return (previousValue & 4294967264) >> 5;
        }

        protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget
        ↪  = left;

        protected override void SetRight(ulong node, ulong right) =>
        ↪  Links[node].RightAsTarget = right;

        protected override void SetSize(ulong node, ulong size)
        {
            var previousValue = Links[node].SizeAsTarget;
            //var modified = MathHelpers.PartialWrite(previousValue, size, 5, -5);
            var modified = (previousValue & 31) | ((size & 134217727) << 5);
            Links[node].SizeAsTarget = modified;
        }

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

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

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

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

        protected override sbyte GetBalance(ulong node)
        {
```

```csharp
                        var previousValue = Links[node].SizeAsTarget;
                        //var value = MathHelpers.PartialRead(previousValue, 0, 3);
                        var value = previousValue & 7;
                        var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                            ↪  124 : value & 3);
                        return unpackedValue;
                    }

                    protected override void SetBalance(ulong node, sbyte value)
                    {
                        var previousValue = Links[node].SizeAsTarget;
                        var packagedValue = (ulong)(((((byte)value >> 5) & 4) | value & 3);
                        //var modified = MathHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
                        var modified = (previousValue & 4294967288) | (packagedValue & 7);
                        Links[node].SizeAsTarget = modified;
                    }

                    protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
                        => Links[first].Target < Links[second].Target ||
                            (Links[first].Target == Links[second].Target && Links[first].Source <
                                ↪  Links[second].Source);

                    protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
                        => Links[first].Target > Links[second].Target ||
                            (Links[first].Target == Links[second].Target && Links[first].Source >
                                ↪  Links[second].Source);

                    protected override ulong GetTreeRoot() => Header->FirstAsTarget;

                    protected override ulong GetBasePartValue(ulong link) => Links[link].Target;

                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    protected override void ClearNode(ulong node)
                    {
                        Links[node].LeftAsTarget = 0UL;
                        Links[node].RightAsTarget = 0UL;
                        Links[node].SizeAsTarget = 0UL;
                    }
                }
            }
        }
    }
```

./Sequences/Converters/BalancedVariantConverter.cs

```csharp
using System.Collections.Generic;

namespace Platform.Data.Doublets.Sequences.Converters
{
    public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }

        public override TLink Convert(IList<TLink> sequence)
        {
            var length = sequence.Count;
            if (length < 1)
            {
                return default;
            }
            if (length == 1)
            {
                return sequence[0];
            }
            // Make copy of next layer
            if (length > 2)
            {
                // TODO: Try to use stackalloc (which at the moment is not working with
                    ↪  generics) but will be possible with Sigil
                var halvedSequence = new TLink[(length / 2) + (length % 2)];
                HalveSequence(halvedSequence, sequence, length);
                sequence = halvedSequence;
                length = halvedSequence.Length;
            }
            // Keep creating layer after layer
            while (length > 2)
            {
                HalveSequence(sequence, sequence, length);
                length = (length / 2) + (length % 2);
            }
            return Links.GetOrCreate(sequence[0], sequence[1]);
```

```csharp
        }

        private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
        {
            var loopedLength = length - (length % 2);
            for (var i = 0; i < loopedLength; i += 2)
            {
                destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
            }
            if (length > loopedLength)
            {
                destination[length / 2] = source[length - 1];
            }
        }
    }
}
```

./Sequences/Converters/CompressingConverter.cs

```csharp
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Collections;
using Platform.Helpers.Singletons;
using Platform.Numbers;
using Platform.Data.Constants;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;

namespace Platform.Data.Doublets.Sequences.Converters
{
    /// <remarks>
    /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
    ///   Links на этапе сжатия.
    ///      А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
    ///   таком случае тип значения элемента массива может быть любым, как char так и ulong.
    ///      Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
    ///   пар, а так же разом выполнить замену.
    /// </remarks>
    public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        private static readonly LinksCombinedConstants<bool, TLink, long> _constants =
            Default<LinksCombinedConstants<bool, TLink, long>>.Instance;
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly IConverter<IList<TLink>, TLink> _baseConverter;
        private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
        private readonly TLink _minFrequencyToCompress;
        private readonly bool _doInitialFrequenciesIncrement;
        private Doublet<TLink> _maxDoublet;
        private LinkFrequency<TLink> _maxDoubletData;

        private struct HalfDoublet
        {
            public TLink Element;
            public LinkFrequency<TLink> DoubletData;

            public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
            {
                Element = element;
                DoubletData = doubletData;
            }

            public override string ToString() => $"{Element}: ({DoubletData})";
        }

        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
            : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
        {
        }

        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
            doInitialFrequenciesIncrement)
            : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
                doInitialFrequenciesIncrement)
        {
        }
```

```csharp
        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
        ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
        ↪  minFrequencyToCompress, bool doInitialFrequenciesIncrement)
            : base(links)
        {
            _baseConverter = baseConverter;
            _doubletFrequenciesCache = doubletFrequenciesCache;
            if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
            {
                minFrequencyToCompress = Integer<TLink>.One;
            }
            _minFrequencyToCompress = minFrequencyToCompress;
            _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
            ResetMaxDoublet();
        }

        public override TLink Convert(IList<TLink> source) =>
        ↪  _baseConverter.Convert(Compress(source));

        /// <remarks>
        /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
        /// Faster version (doublets' frequencies dictionary is not recreated).
        /// </remarks>
        private IList<TLink> Compress(IList<TLink> sequence)
        {
            if (sequence.IsNullOrEmpty())
            {
                return null;
            }
            if (sequence.Count == 1)
            {
                return sequence;
            }
            if (sequence.Count == 2)
            {
                return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
            }
            // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
            var copy = new HalfDoublet[sequence.Count];
            Doublet<TLink> doublet = default;
            for (var i = 1; i < sequence.Count; i++)
            {
                doublet.Source = sequence[i - 1];
                doublet.Target = sequence[i];
                LinkFrequency<TLink> data;
                if (_doInitialFrequenciesIncrement)
                {
                    data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
                }
                else
                {
                    data = _doubletFrequenciesCache.GetFrequency(ref doublet);
                    if (data == null)
                    {
                        throw new NotSupportedException("If you ask not to increment
                        ↪  frequencies, it is expected that all frequencies for the sequence
                        ↪  are prepared.");
                    }
                }
                copy[i - 1].Element = sequence[i - 1];
                copy[i - 1].DoubletData = data;
                UpdateMaxDoublet(ref doublet, data);
            }
            copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
            copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
            if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
            {
                var newLength = ReplaceDoublets(copy);
                sequence = new TLink[newLength];
                for (int i = 0; i < newLength; i++)
                {
                    sequence[i] = copy[i].Element;
                }
            }
            return sequence;
        }

        /// <remarks>
```

```csharp
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) &&
                          ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
150                      {
151                          if (r > 0)
152                          {
153                              var previous = copy[w - 1].Element;
154                              copy[w - 1].DoubletData.DecrementFrequency();
155                              copy[w - 1].DoubletData =
                                  ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
                                  ↪ maxDoubletReplacementLink);
156                          }
157                          if (r < oldLengthMinusTwo)
158                          {
159                              var next = copy[r + 2].Element;
160                              copy[r + 1].DoubletData.DecrementFrequency();
161                              copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma↲
                                  ↪ xDoubletReplacementLink,
                                  ↪ next);
162                          }
163                          copy[w++].Element = maxDoubletReplacementLink;
164                          r++;
165                          newLength--;
166                      }
167                      else
168                      {
169                          copy[w++] = copy[r];
170                      }
171                  }
172                  if (w < newLength)
173                  {
174                      copy[w] = copy[r];
175                  }
176                  oldLength = newLength;
177                  ResetMaxDoublet();
178                  UpdateMaxDoublet(copy, newLength);
179              }
180              return newLength;
181          }
182
183          [MethodImpl(MethodImplOptions.AggressiveInlining)]
184          private void ResetMaxDoublet()
185          {
186              _maxDoublet = new Doublet<TLink>();
187              _maxDoubletData = new LinkFrequency<TLink>();
188          }
189
190          [MethodImpl(MethodImplOptions.AggressiveInlining)]
191          private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
192          {
193              Doublet<TLink> doublet = default;
194              for (var i = 1; i < length; i++)
195              {
196                  doublet.Source = copy[i - 1].Element;
197                  doublet.Target = copy[i].Element;
198                  UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
199              }
200          }
201
```

```
202            [MethodImpl(MethodImplOptions.AggressiveInlining)]
203            private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
204            {
205                var frequency = data.Frequency;
206                var maxFrequency = _maxDoubletData.Frequency;
207                //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
                ↪   (maxFrequency == 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(ArithmeticHelpers.Add(doublet.Source, doublet.Target),
                    ↪   ArithmeticHelpers.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /*
                    ↪   gives better stability and better compression on sequent data and even on
                    ↪   rundom numbers data (but gives collisions anyway) */
210                {
211                    _maxDoublet = doublet;
212                    _maxDoubletData = data;
213                }
214            }
215        }
216    }
```

./Sequences/Converters/LinksListToSequenceConverterBase.cs

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

./Sequences/Converters/OptimalVariantConverter.cs

```
1   using System.Collections.Generic;
2   using System.Linq;
3   using Platform.Interfaces;
4
5   namespace Platform.Data.Doublets.Sequences.Converters
6   {
7       public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
8       {
9           private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪   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,
                                    ↪ currentLevel) :
49                                    i < 2 ?
50                                    GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
51                                    GetGreatestNeigbourLowerThanCurrentOrCurrent(previousLevel,
                                    ↪ currentLevel, levels[i + 1]);
52                                levels[w] = newLevel;
53                                previousLevel = currentLevel;
54                                w++;
55                                levelRepeat = 0;
56                                skipOnce = true;
57                            }
58                            else if (i == length - 1)
59                            {
60                                sequence[w] = sequence[i];
61                                levels[w] = levels[i];
62                                w++;
63                            }
64                        }
65                        else
66                        {
67                            currentLevel = levels[i];
68                            levelRepeat = 1;
69                            if (skipOnce)
70                            {
71                                skipOnce = false;
72                            }
73                            else
74                            {
75                                sequence[w] = sequence[i - 1];
76                                levels[w] = levels[i - 1];
77                                previousLevel = levels[w];
78                                w++;
79                            }
80                            if (i == length - 1)
81                            {
82                                sequence[w] = sequence[i];
83                                levels[w] = levels[i];
84                                w++;
85                            }
86                        }
87                    }
88                    length = w;
89                }
90                return links.GetOrCreate(sequence[0], sequence[1]);
91            }
92
93            private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink
            ↪ current, TLink next)
94            {
95                return _comparer.Compare(previous, next) > 0
96                    ? _comparer.Compare(previous, current) < 0 ? previous : current
97                    : _comparer.Compare(next, current) < 0 ? next : current;
98            }
99
100           private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
            ↪ _comparer.Compare(next, current) < 0 ? next : current;
101
102           private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
            ↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
103       }
104   }
```

./Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs

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

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

./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs

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

./Sequences/CreteriaMatchers/MarkedSequenceCreteriaMatcher.cs

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

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

```csharp
     {
         private static readonly EqualityComparer<TLink> _equalityComparer =
         ↪  EqualityComparer<TLink>.Default;

         private readonly IStack<TLink> _stack;
         private readonly ISequenceHeightProvider<TLink> _heightProvider;

         public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
         ↪  ISequenceHeightProvider<TLink> heightProvider)
             : base(links)
         {
             _stack = stack;
             _heightProvider = heightProvider;
         }

         public TLink Append(TLink sequence, TLink appendant)
         {
             var cursor = sequence;
             while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
             {
                 var source = Links.GetSource(cursor);
                 var target = Links.GetTarget(cursor);
                 if (_equalityComparer.Equals(_heightProvider.Get(source),
                 ↪  _heightProvider.Get(target)))
                 {
                     break;
                 }
                 else
                 {
                     _stack.Push(source);
                     cursor = target;
                 }
             }
             var left = cursor;
             var right = appendant;
             while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
             {
                 right = Links.GetOrCreate(left, right);
                 left = cursor;
             }
             return Links.GetOrCreate(left, right);
         }
     }
 }
```

./Sequences/DuplicateSegmentsCounter.cs

```csharp
using System.Collections.Generic;
using System.Linq;
using Platform.Interfaces;

namespace Platform.Data.Doublets.Sequences
{
    public class DuplicateSegmentsCounter<TLink> : ICounter<int>
    {
        private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
        ↪  _duplicateFragmentsProvider;
        public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
        ↪  IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
        ↪  duplicateFragmentsProvider;
        public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
    }
}
```

./Sequences/DuplicateSegmentsProvider.cs

```csharp
using System;
using System.Linq;
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Collections;
using Platform.Collections.Lists;
using Platform.Collections.Segments;
using Platform.Collections.Segments.Walkers;
using Platform.Helpers;
using Platform.Helpers.Singletons;
using Platform.Numbers;
using Platform.Data.Sequences;

namespace Platform.Data.Doublets.Sequences
{
```

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

```csharp
                return resultList;
        }

        protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
        ↪  length) => new Segment<TLink>(elements, offset, length);

        protected override void OnDublicateFound(Segment<TLink> segment)
        {
                var duplicates = CollectDuplicatesForSegment(segment);
                if (duplicates.Count > 1)
                {
                        _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                        ↪  duplicates));
                }
        }

        private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
        {
                var duplicates = new List<TLink>();
                var readAsElement = new HashSet<TLink>();
                _sequences.Each(sequence =>
                {
                        duplicates.Add(sequence);
                        readAsElement.Add(sequence);
                        return true; // Continue
                }, segment);
                if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
                {
                        return new List<TLink>();
                }
                foreach (var duplicate in duplicates)
                {
                        var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
                        _visited.Set(duplicateBitIndex);
                }
                if (_sequences is Sequences sequencesExperiments)
                {
                        var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
                        ↪  ashSet<ulong>)(object)readAsElement,
                        ↪  (IList<ulong>)segment);
                        foreach (var partiallyMatchedSequence in partiallyMatched)
                        {
                                TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
                                duplicates.Add(sequenceIndex);
                        }
                }
                duplicates.Sort();
                return duplicates;
        }

        private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
        {
                if (!(_links is ILinks<ulong> ulongLinks))
                {
                        return;
                }
                var duplicatesKey = duplicatesItem.Key;
                var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
                Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
                var duplicatesList = duplicatesItem.Value;
                for (int i = 0; i < duplicatesList.Count; i++)
                {
                        ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
                        var formatedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
                        ↪  Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
                        ↪  UnicodeMap.IsCharLink(link.Index) ?
                        ↪  sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
                        Console.WriteLine(formatedSequenceStructure);
                        var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
                        ↪  ulongLinks);
                        Console.WriteLine(sequenceString);
                }
                Console.WriteLine();
        }
    }
}
```

```
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  }
```

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

```
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)]
```

```csharp
public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
{
    _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
    return data;
}

public void IncrementFrequencies(IList<TLink> sequence)
{
    for (var i = 1; i < sequence.Count; i++)
    {
        IncrementFrequency(sequence[i - 1], sequence[i]);
    }
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
{
    var doublet = new Doublet<TLink>(source, target);
    return IncrementFrequency(ref doublet);
}

public void PrintFrequencies(IList<TLink> sequence)
{
    for (var i = 1; i < sequence.Count; i++)
    {
        PrintFrequency(sequence[i - 1], sequence[i]);
    }
}

public void PrintFrequency(TLink source, TLink target)
{
    var number = GetFrequency(source, target).Frequency;
    Console.WriteLine("({0},{1}) - {2}", source, target, number);
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
{
    if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
    {
        data.IncrementFrequency();
    }
    else
    {
        var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
        data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
        if (!_equalityComparer.Equals(link, default))
        {
            data.Frequency = ArithmeticHelpers.Add(data.Frequency,
                _frequencyCounter.Count(link));
        }
        _doubletsCache.Add(doublet, data);
    }
    return data;
}

public void ValidateFrequencies()
{
    foreach (var entry in _doubletsCache)
    {
        var value = entry.Value;
        var linkIndex = value.Link;
        if (!_equalityComparer.Equals(linkIndex, default))
        {
            var frequency = value.Frequency;
            var count = _frequencyCounter.Count(linkIndex);
            // TODO: Why `frequency` always greater than `count` by 1?
            if (((_comparer.Compare(frequency, count) > 0) &&
                (_comparer.Compare(ArithmeticHelpers.Subtract(frequency, count),
                Integer<TLink>.One) > 0))
                || ((_comparer.Compare(count, frequency) > 0) &&
                (_comparer.Compare(ArithmeticHelpers.Subtract(count, frequency),
                Integer<TLink>.One) > 0)))
            {
                throw new InvalidOperationException("Frequencies validation failed.");
            }
        }
        //else
```

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

./Sequences/Frequencies/Cache/LinkFrequency.cs

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

./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs

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

./Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs

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

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

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

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

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

./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    }
```

./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 = ArithmeticHelpers.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  }
```

./Sequences/HeightProviders/CachedSequenceHeightProvider.cs

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

./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

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

```
 7      {
 8          private readonly ICreteriaMatcher<TLink> _elementMatcher;
 9
10          public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICreteriaMatcher<TLink>
   ↪   elementMatcher) : base(links) => _elementMatcher = elementMatcher;
11
12          public TLink Get(TLink sequence)
13          {
14              var height = default(TLink);
15              var pairOrElement = sequence;
16              while (!_elementMatcher.IsMatched(pairOrElement))
17              {
18                  pairOrElement = Links.GetTarget(pairOrElement);
19                  height = ArithmeticHelpers.Increment(height);
20              }
21              return height;
22          }
23      }
24  }
```

./Sequences/HeightProviders/ISequenceHeightProvider.cs

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

./Sequences/Sequences.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Runtime.CompilerServices;
 5  using Platform.Collections;
 6  using Platform.Collections.Lists;
 7  using Platform.Threading.Synchronization;
 8  using Platform.Helpers.Singletons;
 9  using LinkIndex = System.UInt64;
10  using Platform.Data.Constants;
11  using Platform.Data.Sequences;
12  using Platform.Data.Doublets.Sequences.Walkers;
13
14  namespace Platform.Data.Doublets.Sequences
15  {
16      /// <summary>
17      /// Представляет коллекцию последовательностей связей.
18      /// </summary>
19      /// <remarks>
20      /// Обязательно реализовать атомарность каждого публичного метода.
21      ///
22      /// TODO:
23      ///
24      /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
25      /// через естественную группировку по unicode типам, все whitespace вместе, все символы
   ↪   вместе, все числа вместе и т.п.
26      /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
   ↪   графа)
27      ///
28      /// x*y - найти все связи между, в последовательностях любой формы, если не стоит
   ↪   ограничитель на то, что является последовательностью, а что нет,
29      /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
   ↪   порядке.
30      ///
31      /// Рост последовательности слева и справа.
32      /// Поиск со звёздочкой.
33      /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
34      /// так же проблема может быть решена при реализации дистанционных триггеров.
35      /// Нужны ли уникальные указатели вообще?
36      /// Что если обращение к информации будет происходить через содержимое всегда?
37      ///
38      /// Писать тесты.
39      ///
40      ///
41      /// Можно убрать зависимость от конкретной реализации Links,
42      /// на зависимость от абстрактного элемента, который может быть представлен несколькими
   ↪   способами.
43      ///
```

```csharp
    /// Можно ли как-то сделать один общий интерфейс
    ///
    ///
    /// Блокчейн и/или гит для распределённой записи транзакций.
    ///
    /// </remarks>
    public partial class Sequences : ISequences<ulong> // IList<string>, IList<ulong[]> (после
    ↪  завершения реализации Sequences)
    {
        private static readonly LinksCombinedConstants<bool, ulong, long> _constants =
        ↪  Default<LinksCombinedConstants<bool, ulong, long>>.Instance;

        /// <summary>Возвращает значение ulong, обозначающее любое количество связей.</summary>
        public const ulong ZeroOrMany = ulong.MaxValue;

        public SequencesOptions<ulong> Options;
        public readonly SynchronizedLinks<ulong> Links;
        public readonly ISynchronization Sync;

        public Sequences(SynchronizedLinks<ulong> links)
            : this(links, new SequencesOptions<ulong>())
        {
        }

        public Sequences(SynchronizedLinks<ulong> links, SequencesOptions<ulong> options)
        {
            Links = links;
            Sync = links.SyncRoot;
            Options = options;

            Options.ValidateOptions();
            Options.InitOptions(Links);
        }

        public bool IsSequence(ulong sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                if (Options.UseSequenceMarker)
                {
                    return Options.MarkedSequenceMatcher.IsMatched(sequence);
                }
                return !Links.Unsync.IsPartialPoint(sequence);
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private ulong GetSequenceByElements(ulong sequence)
        {
            if (Options.UseSequenceMarker)
            {
                return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
            }
            return sequence;
        }

        private ulong GetSequenceElements(ulong sequence)
        {
            if (Options.UseSequenceMarker)
            {
                var linkContents = new UInt64Link(Links.GetLink(sequence));
                if (linkContents.Source == Options.SequenceMarkerLink)
                {
                    return linkContents.Target;
                }
                if (linkContents.Target == Options.SequenceMarkerLink)
                {
                    return linkContents.Source;
                }
            }
            return sequence;
        }

        #region Count

        public ulong Count(params ulong[] sequence)
        {
            if (sequence.Length == 0)
            {
```

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

```csharp
200                }
201                else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
202                {
203                    return CompactCore(sequence);
204                }
205                if (sequenceRoot == default)
206                {
207                    sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
208                }
209                if (Options.UseSequenceMarker)
210                {
211                    Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
212                }
213                return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
214            }

215
216            #endregion

217
218            #region Each

219
220            public List<ulong> Each(params ulong[] sequence)
221            {
222                var results = new List<ulong>();
223                Each(results.AddAndReturnTrue, sequence);
224                return results;
225            }

226
227            public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
228            {
229                return Sync.ExecuteReadOperation(() =>
230                {
231                    if (sequence.IsNullOrEmpty())
232                    {
233                        return true;
234                    }
235                    Links.EnsureEachLinkIsAnyOrExists(sequence);
236                    if (sequence.Count == 1)
237                    {
238                        var link = sequence[0];
239                        if (link == _constants.Any)
240                        {
241                            return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
242                        }
243                        return handler(link);
244                    }
245                    if (sequence.Count == 2)
246                    {
247                        return Links.Unsync.Each(sequence[0], sequence[1], handler);
248                    }
249                    if (Options.UseIndex && !Options.Indexer.CheckIndex(sequence))
250                    {
251                        return false;
252                    }
253                    return EachCore(handler, sequence);
254                });
255            }

256
257            private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
258            {
259                var matcher = new Matcher(this, sequence, new HashSet<LinkIndex>(), handler);
260                // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
                //   Id.
261                Func<ulong, bool> innerHandler = Options.UseSequenceMarker ? (Func<ulong,
                //   bool>)matcher.HandleFullMatchedSequence : matcher.HandleFullMatched;
262                //if (sequence.Length >= 2)
263                if (!StepRight(innerHandler, sequence[0], sequence[1]))
264                {
265                    return false;
266                }
267                var last = sequence.Count - 2;
268                for (var i = 1; i < last; i++)
269                {
270                    if (!PartialStepRight(innerHandler, sequence[i], sequence[i + 1]))
271                    {
272                        return false;
273                    }
274                }
275                if (sequence.Count >= 3)
276                {
```

```csharp
                    if (!StepLeft(innerHandler, sequence[sequence.Count - 2],
                    ↪  sequence[sequence.Count - 1]))
                    {
                        return false;
                    }
                }
                return true;
            }

            private bool PartialStepRight(Func<ulong, bool> handler, ulong left, ulong right)
            {
                return Links.Unsync.Each(_constants.Any, left, doublet =>
                {
                    if (!StepRight(handler, doublet, right))
                    {
                        return false;
                    }
                    if (left != doublet)
                    {
                        return PartialStepRight(handler, doublet, right);
                    }
                    return true;
                });
            }

            private bool StepRight(Func<ulong, bool> handler, ulong left, ulong right) =>
            ↪  Links.Unsync.Each(left, _constants.Any, rightStep => TryStepRightUp(handler, right,
            ↪  rightStep));

            private bool TryStepRightUp(Func<ulong, bool> handler, ulong right, ulong stepFrom)
            {
                var upStep = stepFrom;
                var firstSource = Links.Unsync.GetTarget(upStep);
                while (firstSource != right && firstSource != upStep)
                {
                    upStep = firstSource;
                    firstSource = Links.Unsync.GetSource(upStep);
                }
                if (firstSource == right)
                {
                    return handler(stepFrom);
                }
                return true;
            }

            private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong right) =>
            ↪  Links.Unsync.Each(_constants.Any, right, leftStep => TryStepLeftUp(handler, left,
            ↪  leftStep));

            private bool TryStepLeftUp(Func<ulong, bool> handler, ulong left, ulong stepFrom)
            {
                var upStep = stepFrom;
                var firstTarget = Links.Unsync.GetSource(upStep);
                while (firstTarget != left && firstTarget != upStep)
                {
                    upStep = firstTarget;
                    firstTarget = Links.Unsync.GetTarget(upStep);
                }
                if (firstTarget == left)
                {
                    return handler(stepFrom);
                }
                return true;
            }

            #endregion

            #region Update

            public ulong Update(ulong[] sequence, ulong[] newSequence)
            {
                if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
                {
                    return _constants.Null;
                }
                if (sequence.IsNullOrEmpty())
                {
                    return Create(newSequence);
                }
```

```csharp
            if (newSequence.IsNullOrEmpty())
            {
                Delete(sequence);
                return _constants.Null;
            }
            return Sync.ExecuteWriteOperation(() =>
            {
                Links.EnsureEachLinkIsAnyOrExists(sequence);
                Links.EnsureEachLinkExists(newSequence);
                return UpdateCore(sequence, newSequence);
            });
        }

        private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
        {
            ulong bestVariant;
            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
              !sequence.EqualTo(newSequence))
            {
                bestVariant = CompactCore(newSequence);
            }
            else
            {
                bestVariant = CreateCore(newSequence);
            }
            // TODO: Check all options only ones before loop execution
            // Возможно нужно две версии Each, возвращающий фактические последовательности и с
              маркером,
            // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
              можно получить имея только фактические последовательности.
            foreach (var variant in Each(sequence))
            {
                if (variant != bestVariant)
                {
                    UpdateOneCore(variant, bestVariant);
                }
            }
            return bestVariant;
        }

        private void UpdateOneCore(ulong sequence, ulong newSequence)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(sequence);
                var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                var newSequenceElements = GetSequenceElements(newSequence);
                var newSequenceLink = GetSequenceByElements(newSequenceElements);
                if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
                {
                    if (sequenceLink != _constants.Null)
                    {
                        Links.Unsync.Merge(sequenceLink, newSequenceLink);
                    }
                    Links.Unsync.Merge(sequenceElements, newSequenceElements);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(sequence);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    var newSequenceElements = GetSequenceElements(newSequence);
                    var newSequenceLink = GetSequenceByElements(newSequenceElements);
                    if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
                    {
                        if (sequenceLink != _constants.Null)
                        {
                            Links.Unsync.Merge(sequenceLink, newSequenceLink);
                        }
                        Links.Unsync.Merge(sequenceElements, newSequenceElements);
                    }
                    else
```

```csharp
                    {
                        if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
                        {
                            Links.Unsync.Merge(sequence, newSequence);
                        }
                    }
                }
            }
        }

        #endregion

        #region Delete

        public void Delete(params ulong[] sequence)
        {
            Sync.ExecuteWriteOperation(() =>
            {
                // TODO: Check all options only ones before loop execution
                foreach (var linkToDelete in Each(sequence))
                {
                    DeleteOneCore(linkToDelete);
                }
            });
        }

        private void DeleteOneCore(ulong link)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(link);
                var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                if (Options.UseCascadeDelete || CountReferences(link) == 0)
                {
                    if (sequenceLink != _constants.Null)
                    {
                        Links.Unsync.Delete(sequenceLink);
                    }
                    Links.Unsync.Delete(link);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(link);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    if (Options.UseCascadeDelete || CountReferences(link) == 0)
                    {
                        if (sequenceLink != _constants.Null)
                        {
                            Links.Unsync.Delete(sequenceLink);
                        }
                        Links.Unsync.Delete(link);
                    }
                }
                else
                {
                    if (Options.UseCascadeDelete || CountReferences(link) == 0)
                    {
                        Links.Unsync.Delete(link);
                    }
                }
            }
        }

        #endregion

        #region Compactification

        /// <remarks>
        /// bestVariant можно выбирать по максимальному числу использований,
        /// но балансированный позволяет гарантировать уникальность (если есть возможность,
        /// гарантировать его использование в других местах).
        ///
        /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
        /// </remarks>
```

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

```csharp
                _stopableHandler = stopableHandler;
                _readAsElements = readAsElements;
            }

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

            public bool FullMatch(LinkIndex sequenceToMatch)
            {
                _filterPosition = 0;
                foreach (var part in Walk(sequenceToMatch))
                {
                    if (!FullMatchCore(Links.GetIndex(part)))
                    {
                        break;
                    }
                }
                return _filterPosition == _patternSequence.Count;
            }

            private bool FullMatchCore(LinkIndex element)
            {
                if (_filterPosition == _patternSequence.Count)
                {
                    _filterPosition = -2; // Длиннее чем нужно
                    return false;
                }
                if (_patternSequence[_filterPosition] != _constants.Any
                 && element != _patternSequence[_filterPosition])
                {
                    _filterPosition = -1;
                    return false; // Начинается/Продолжается иначе
                }
                _filterPosition++;
                return true;
            }

            public void AddFullMatchedToResults(ulong sequenceToMatch)
            {
                if (FullMatch(sequenceToMatch))
                {
                    _results.Add(sequenceToMatch);
                }
            }

            public bool HandleFullMatched(ulong sequenceToMatch)
            {
                if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
                {
                    return _stopableHandler(sequenceToMatch);
                }
                return true;
            }

            public bool HandleFullMatchedSequence(ulong sequenceToMatch)
            {
                var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
                if (sequence != _constants.Null && FullMatch(sequenceToMatch) &&
                ↪   _results.Add(sequenceToMatch))
                {
                    return _stopableHandler(sequence);
                }
                return true;
            }

            /// <remarks>
            /// TODO: Add support for LinksConstants.Any
            /// </remarks>
            public bool PartialMatch(LinkIndex sequenceToMatch)
            {
                _filterPosition = -1;
                foreach (var part in Walk(sequenceToMatch))
                {
                    if (!PartialMatchCore(Links.GetIndex(part)))
                    {
                        break;
                    }
                }
```

```csharp
                     return _filterPosition == _patternSequence.Count - 1;
                }

                private bool PartialMatchCore(LinkIndex element)
                {
                    if (_filterPosition == (_patternSequence.Count - 1))
                    {
                        return false; // Нашлось
                    }
                    if (_filterPosition >= 0)
                    {
                        if (element == _patternSequence[_filterPosition + 1])
                        {
                            _filterPosition++;
                        }
                        else
                        {
                            _filterPosition = -1;
                        }
                    }
                    if (_filterPosition < 0)
                    {
                        if (element == _patternSequence[0])
                        {
                            _filterPosition = 0;
                        }
                    }
                    return true; // Ищем дальше
                }

                public void AddPartialMatchedToResults(ulong sequenceToMatch)
                {
                    if (PartialMatch(sequenceToMatch))
                    {
                        _results.Add(sequenceToMatch);
                    }
                }

                public bool HandlePartialMatched(ulong sequenceToMatch)
                {
                    if (PartialMatch(sequenceToMatch))
                    {
                        return _stopableHandler(sequenceToMatch);
                    }
                    return true;
                }

                public void AddAllPartialMatchedToResults(IEnumerable<ulong> sequencesToMatch)
                {
                    foreach (var sequenceToMatch in sequencesToMatch)
                    {
                        if (PartialMatch(sequenceToMatch))
                        {
                            _results.Add(sequenceToMatch);
                        }
                    }
                }

                public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<ulong>
                    sequencesToMatch)
                {
                    foreach (var sequenceToMatch in sequencesToMatch)
                    {
                        if (PartialMatch(sequenceToMatch))
                        {
                            _readAsElements.Add(sequenceToMatch);
                            _results.Add(sequenceToMatch);
                        }
                    }
                }
            }

        #endregion
        }
    }
```

./Sequences/Sequences.Experiments.cs
```csharp
using System;
using LinkIndex = System.UInt64;
```

```csharp
using System.Collections.Generic;
using Stack = System.Collections.Generic.Stack<ulong>;
using System.Linq;
using System.Text;
using Platform.Collections;
using Platform.Numbers;
using Platform.Data.Exceptions;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Walkers;

namespace Platform.Data.Doublets.Sequences
{
    partial class Sequences
    {
        #region Create All Variants (Not Practical)

        /// <remarks>
        /// Number of links that is needed to generate all variants for
        /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
        /// </remarks>
        public ulong[] CreateAllVariants2(ulong[] sequence)
        {
            return Sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return new ulong[0];
                }
                Links.EnsureEachLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return sequence;
                }
                return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
            });
        }

        private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
        {
#if DEBUG
            if ((stopAt - startAt) < 0)
            {
                throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
                    меньше или равен stopAt");
            }
#endif
            if ((stopAt - startAt) == 0)
            {
                return new[] { sequence[startAt] };
            }
            if ((stopAt - startAt) == 1)
            {
                return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt], sequence[stopAt])
                    };
            }
            var variants = new ulong[(ulong)MathHelpers.Catalan(stopAt - startAt)];
            var last = 0;
            for (var splitter = startAt; splitter < stopAt; splitter++)
            {
                var left = CreateAllVariants2Core(sequence, startAt, splitter);
                var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
                for (var i = 0; i < left.Length; i++)
                {
                    for (var j = 0; j < right.Length; j++)
                    {
                        var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
                        if (variant == _constants.Null)
                        {
                            throw new NotImplementedException("Creation cancellation is not
                                implemented.");
                        }
                        variants[last++] = variant;
                    }
                }
            }
            return variants;
        }
```

```csharp
        public List<ulong> CreateAllVariants1(params ulong[] sequence)
        {
            return Sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return new List<ulong>();
                }
                Links.Unsync.EnsureEachLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return new List<ulong> { sequence[0] };
                }
                var results = new List<ulong>((int)MathHelpers.Catalan(sequence.Length));
                return CreateAllVariants1Core(sequence, results);
            });
        }

        private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
        {
            if (sequence.Length == 2)
            {
                var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
                if (link == _constants.Null)
                {
                    throw new NotImplementedException("Creation cancellation is not
                    ↪  implemented.");
                }
                results.Add(link);
                return results;
            }
            var innerSequenceLength = sequence.Length - 1;
            var innerSequence = new ulong[innerSequenceLength];
            for (var li = 0; li < innerSequenceLength; li++)
            {
                var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
                if (link == _constants.Null)
                {
                    throw new NotImplementedException("Creation cancellation is not
                    ↪  implemented.");
                }
                for (var isi = 0; isi < li; isi++)
                {
                    innerSequence[isi] = sequence[isi];
                }
                innerSequence[li] = link;
                for (var isi = li + 1; isi < innerSequenceLength; isi++)
                {
                    innerSequence[isi] = sequence[isi + 1];
                }
                CreateAllVariants1Core(innerSequence, results);
            }
            return results;
        }

        #endregion

        public HashSet<ulong> Each1(params ulong[] sequence)
        {
            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
            Each1(link =>
            {
                if (!visitedLinks.Contains(link))
                {
                    visitedLinks.Add(link); // изучить почему случаются повторы
                }
                return true;
            }, sequence);
            return visitedLinks;
        }

        private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
        {
            if (sequence.Length == 2)
            {
                Links.Unsync.Each(sequence[0], sequence[1], handler);
            }
            else
```

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

```csharp
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 =>
```

```csharp
            {
                StepLeft(handler, left, doublet);
                if (right != doublet)
                {
                    PartialStepLeft(handler, left, doublet);
                }
                return true;
            });
        }

        private void StepLeft(Action<ulong> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(_constants.Any, right, leftStep =>
            {
                TryStepLeftUp(handler, left, leftStep);
                return true;
            });
        }

        private void TryStepLeftUp(Action<ulong> handler, ulong left, ulong stepFrom)
        {
            var upStep = stepFrom;
            var firstTarget = Links.Unsync.GetSource(upStep);
            while (firstTarget != left && firstTarget != upStep)
            {
                upStep = firstTarget;
                firstTarget = Links.Unsync.GetTarget(upStep);
            }
            if (firstTarget == left)
            {
                handler(stepFrom);
            }
        }

        private bool StartsWith(ulong sequence, ulong link)
        {
            var upStep = sequence;
            var firstSource = Links.Unsync.GetSource(upStep);
            while (firstSource != link && firstSource != upStep)
            {
                upStep = firstSource;
                firstSource = Links.Unsync.GetSource(upStep);
            }
            return firstSource == link;
        }

        private bool EndsWith(ulong sequence, ulong link)
        {
            var upStep = sequence;
            var lastTarget = Links.Unsync.GetTarget(upStep);
            while (lastTarget != link && lastTarget != upStep)
            {
                upStep = lastTarget;
                lastTarget = Links.Unsync.GetTarget(upStep);
            }
            return lastTarget == link;
        }

        public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                var results = new List<ulong>();
                if (sequence.Length > 0)
                {
                    Links.EnsureEachLinkExists(sequence);
                    var firstElement = sequence[0];
                    if (sequence.Length == 1)
                    {
                        results.Add(firstElement);
                        return results;
                    }
                    if (sequence.Length == 2)
                    {
                        var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
                        if (doublet != _constants.Null)
                        {
                            results.Add(doublet);
                        }
```

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

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

```
529             var sb = new StringBuilder();
530             sb.Append('{');
531             if (links.Exists(sequenceLink))
532             {
533                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
534                     x => linksInSequence.Contains(x) || links.IsFullPoint(x),
                    ↪  entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
535                 {
536                     if (insertComma && sb.Length > 1)
537                     {
538                         sb.Append(',');
539                     }
540                     if (entered.Contains(element))
541                     {
542                         sb.Append('{');
543                         elementToString(sb, element);
544                         sb.Append('}');
545                     }
546                     else
547                     {
548                         elementToString(sb, element);
549                     }
550                     if (sb.Length < MaxSequenceFormatSize)
551                     {
552                         return true;
553                     }
554                     sb.Append(insertComma ? ", ..." : "...");
555                     return false;
556                 });
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,
                        ↪  Links.Unsync.GetTarget,
580                             x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                            ↪  x =>
581                         {
582                             if (filterPosition == (sequence.Length - 1))
583                             {
584                                 return false;
585                             }
586                             if (filterPosition >= 0)
587                             {
588                                 if (x == sequence[filterPosition + 1])
589                                 {
590                                     filterPosition++;
591                                 }
592                                 else
593                                 {
594                                     return false;
595                                 }
596                             }
597                             if (filterPosition < 0)
598                             {
599                                 if (x == sequence[0])
600                                 {
601                                     filterPosition = 0;
602                                 }
603                             }
```

```csharp
                                return true;
                        });
                    if (filterPosition == (sequence.Length - 1))
                    {
                        filteredResults.Add(result);
                    }
                }
                return filteredResults;
            }
            return new List<ulong>();
        });
    }

    public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
    {
        return Sync.ExecuteReadOperation(() =>
        {
            if (sequence.Length > 0)
            {
                Links.EnsureEachLinkExists(sequence);
                var results = new HashSet<ulong>();
                for (var i = 0; i < sequence.Length; i++)
                {
                    AllUsagesCore(sequence[i], results);
                }
                var filteredResults = new HashSet<ulong>();
                var matcher = new Matcher(this, sequence, filteredResults, null);
                matcher.AddAllPartialMatchedToResults(results);
                return filteredResults;
            }
            return new HashSet<ulong>();
        });
    }

    public bool GetAllPartiallyMatchingSequences2(Func<ulong, bool> handler, params ulong[]
    ↪  sequence)
    {
        return Sync.ExecuteReadOperation(() =>
        {
            if (sequence.Length > 0)
            {
                Links.EnsureEachLinkExists(sequence);

                var results = new HashSet<ulong>();
                var filteredResults = new HashSet<ulong>();
                var matcher = new Matcher(this, sequence, filteredResults, handler);
                for (var i = 0; i < sequence.Length; i++)
                {
                    if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
                    {
                        return false;
                    }
                }
                return true;
            }
            return true;
        });
    }

    //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
    //{
    //    return Sync.ExecuteReadOperation(() =>
    //    {
    //        if (sequence.Length > 0)
    //        {
    //            _links.EnsureEachLinkIsAnyOrExists(sequence);

    //            var firstResults = new HashSet<ulong>();
    //            var lastResults = new HashSet<ulong>();

    //            var first = sequence.First(x => x != LinksConstants.Any);
    //            var last = sequence.Last(x => x != LinksConstants.Any);

    //            AllUsagesCore(first, firstResults);
    //            AllUsagesCore(last, lastResults);

    //            firstResults.IntersectWith(lastResults);

    //            //for (var i = 0; i < sequence.Length; i++)
```

```csharp
682     //                //    AllUsagesCore(sequence[i], results);
683
684     //                var filteredResults = new HashSet<ulong>();
685     //                var matcher = new Matcher(this, sequence, filteredResults, null);
686     //                matcher.AddAllPartialMatchedToResults(firstResults);
687     //                return filteredResults;
688     //            }
689
690     //            return new HashSet<ulong>();
691     //    });
692     //}

693
694     public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
695     {
696         return Sync.ExecuteReadOperation(() =>
697         {
698             if (sequence.Length > 0)
699             {
700                 Links.EnsureEachLinkIsAnyOrExists(sequence);
701                 var firstResults = new HashSet<ulong>();
702                 var lastResults = new HashSet<ulong>();
703                 var first = sequence.First(x => x != _constants.Any);
704                 var last = sequence.Last(x => x != _constants.Any);
705                 AllUsagesCore(first, firstResults);
706                 AllUsagesCore(last, lastResults);
707                 firstResults.IntersectWith(lastResults);
708                 //for (var i = 0; i < sequence.Length; i++)
709                 //    AllUsagesCore(sequence[i], results);
710                 var filteredResults = new HashSet<ulong>();
711                 var matcher = new Matcher(this, sequence, filteredResults, null);
712                 matcher.AddAllPartialMatchedToResults(firstResults);
713                 return filteredResults;
714             }
715             return new HashSet<ulong>();
716         });
717     }

718
719     public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
    ↪  IList<ulong> sequence)
720     {
721         return Sync.ExecuteReadOperation(() =>
722         {
723             if (sequence.Count > 0)
724             {
725                 Links.EnsureEachLinkExists(sequence);
726                 var results = new HashSet<LinkIndex>();
727                 //var nextResults = new HashSet<ulong>();
728                 //for (var i = 0; i < sequence.Length; i++)
729                 //{
730                 //    AllUsagesCore(sequence[i], nextResults);
731                 //    if (results.IsNullOrEmpty())
732                 //    {
733                 //        results = nextResults;
734                 //        nextResults = new HashSet<ulong>();
735                 //    }
736                 //    else
737                 //    {
738                 //        results.IntersectWith(nextResults);
739                 //        nextResults.Clear();
740                 //    }
741                 //}
742                 var collector1 = new AllUsagesCollector1(Links.Unsync, results);
743                 collector1.Collect(Links.Unsync.GetLink(sequence[0]));
744                 var next = new HashSet<ulong>();
745                 for (var i = 1; i < sequence.Count; i++)
746                 {
747                     var collector = new AllUsagesCollector1(Links.Unsync, next);
748                     collector.Collect(Links.Unsync.GetLink(sequence[i]));
749
750                     results.IntersectWith(next);
751                     next.Clear();
752                 }
753                 var filteredResults = new HashSet<ulong>();
754                 var matcher = new Matcher(this, sequence, filteredResults, null,
    ↪  readAsElements);
755                 matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
    ↪  x)); // OrderBy is a Hack
756                 return filteredResults;
```

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

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

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

```csharp
//////        return results;
//////    if (matches.Count == 2)
//////    {
//////        var merged = new List<ulong>();
//////        for (var j = 0; j < matches[0].Count; j++)
//////            for (var k = 0; k < matches[1].Count; k++)
//////                CloseInnerConnections(merged.Add, matches[0][j],
    ↪ matches[1][k]);
//////        if (merged.Count > 0)
//////            matches = new List<List<ulong>> { merged };
//////        else
//////            return new List<ulong>();
//////    }
//////}
//////if (matches.Count > 0)
//////{
//////    var usages = new HashSet<ulong>();
//////    for (int i = 0; i < sequence.Length; i++)
//////    {
//////        AllUsagesCore(sequence[i], usages);
//////    }
//////    //for (int i = 0; i < matches[0].Count; i++)
//////    //    AllUsagesCore(matches[0][i], usages);
//////    //usages.UnionWith(matches[0]);
//////    return usages.ToList();
//////}
                var firstLinkUsages = new HashSet<ulong>();
                AllUsagesCore(sequence[0], firstLinkUsages);
                firstLinkUsages.Add(sequence[0]);
                //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
    ↪ sequence[0] }; // or all sequences, containing this element?
                //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
    ↪ 1).ToList();
                var results = new HashSet<ulong>();
                foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
    ↪ firstLinkUsages, 1))
                {
                    AllUsagesCore(match, results);
                }
                return results.ToList();
            }
            return new List<ulong>();
        });
    }

    /// <remarks>
    /// TODO: Может потробоваться ограничение на уровень глубины рекурсии
    /// </remarks>
    public HashSet<ulong> AllUsages(ulong link)
    {
        return Sync.ExecuteReadOperation(() =>
        {
            var usages = new HashSet<ulong>();
            AllUsagesCore(link, usages);
            return usages;
        });
    }

    // При сборе всех использований (последовательностей) можно сохранять обратный путь к
    ↪ той связи с которой начинался поиск (STTTSSSTT),
    // причём достаточно одного бита для хранения перехода влево или вправо
    private void AllUsagesCore(ulong link, HashSet<ulong> usages)
    {
        bool handler(ulong doublet)
        {
            if (usages.Add(doublet))
            {
                AllUsagesCore(doublet, usages);
            }
            return true;
        }
        Links.Unsync.Each(link, _constants.Any, handler);
        Links.Unsync.Each(_constants.Any, link, handler);
    }

    public HashSet<ulong> AllBottomUsages(ulong link)
    {
```

```csharp
            return Sync.ExecuteReadOperation(() =>
            {
                var visits = new HashSet<ulong>();
                var usages = new HashSet<ulong>();
                AllBottomUsagesCore(link, visits, usages);
                return usages;
            });
        }

        private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
        ↪   usages)
        {
            bool handler(ulong doublet)
            {
                if (visits.Add(doublet))
                {
                    AllBottomUsagesCore(doublet, visits, usages);
                }
                return true;
            }
            if (Links.Unsync.Count(_constants.Any, link) == 0)
            {
                usages.Add(link);
            }
            else
            {
                Links.Unsync.Each(link, _constants.Any, handler);
                Links.Unsync.Each(_constants.Any, link, handler);
            }
        }

        public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
        {
            if (Options.UseSequenceMarker)
            {
                var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                    ↪   Options.MarkedSequenceMatcher, symbol);
                return counter.Count();
            }
            else
            {
                var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                    ↪   symbol);
                return counter.Count();
            }
        }

        private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool>
        ↪   outerHandler)
        {
            bool handler(ulong doublet)
            {
                if (usages.Add(doublet))
                {
                    if (!outerHandler(doublet))
                    {
                        return false;
                    }
                    if (!AllUsagesCore1(doublet, usages, outerHandler))
                    {
                        return false;
                    }
                }
                return true;
            }
            return Links.Unsync.Each(link, _constants.Any, handler)
                && Links.Unsync.Each(_constants.Any, link, handler);
        }

        public void CalculateAllUsages(ulong[] totals)
        {
            var calculator = new AllUsagesCalculator(Links, totals);
            calculator.Calculate();
        }

        public void CalculateAllUsages2(ulong[] totals)
        {
            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,
    ↪  CalculateCore);
990
991             private bool CalculateCore(ulong link)
992             {
993                 if (_totals[link] == 0)
994                 {
995                     var total = 1UL;
996                     _totals[link] = total;
997                     var visitedChildren = new HashSet<ulong>();
998                     bool linkCalculator(ulong child)
999                     {
1000                        if (link != child && visitedChildren.Add(child))
1001                        {
1002                            total += _totals[child] == 0 ? 1 : _totals[child];
1003                        }
1004                        return true;
1005                    }
1006                    _links.Unsync.Each(link, _constants.Any, linkCalculator);
1007                    _links.Unsync.Each(_constants.Any, link, linkCalculator);
1008                    _totals[link] = total;
1009                }
1010                return true;
1011            }
1012        }
1013
1014        private class AllUsagesCalculator2
1015        {
1016            private readonly SynchronizedLinks<ulong> _links;
1017            private readonly ulong[] _totals;
1018
1019            public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1020            {
1021                _links = links;
1022                _totals = totals;
1023            }
1024
1025            public void Calculate() => _links.Each(_constants.Any, _constants.Any,
    ↪  CalculateCore);
1026
1027            private bool IsElement(ulong link)
1028            {
1029                //_linksInSequence.Contains(link) ||
1030                return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
    ↪  link;
1031            }
1032
1033            private bool CalculateCore(ulong link)
1034            {
1035                // TODO: Проработать защиту от зацикливания
1036                // Основано на SequenceWalker.WalkLeft
1037                Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1038                Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1039                Func<ulong, bool> isElement = IsElement;
1040                void visitLeaf(ulong parent)
1041                {
1042                    if (link != parent)
1043                    {
1044                        _totals[parent]++;
1045                    }
1046                }
1047                void visitNode(ulong parent)
1048                {
1049                    if (link != parent)
1050                    {
```

```csharp
                                _totals[parent]++;
                            }
                        }
                    var stack = new Stack();
                    var element = link;
                    if (isElement(element))
                    {
                        visitLeaf(element);
                    }
                    else
                    {
                        while (true)
                        {
                            if (isElement(element))
                            {
                                if (stack.Count == 0)
                                {
                                    break;
                                }
                                element = stack.Pop();
                                var source = getSource(element);
                                var target = getTarget(element);
                                // Обработка элемента
                                if (isElement(target))
                                {
                                    visitLeaf(target);
                                }
                                if (isElement(source))
                                {
                                    visitLeaf(source);
                                }
                                element = source;
                            }
                            else
                            {
                                stack.Push(element);
                                visitNode(element);
                                element = getTarget(element);
                            }
                        }
                    }
                    _totals[link]++;
                    return true;
                }
            }

        private class AllUsagesCollector
        {
            private readonly ILinks<ulong> _links;
            private readonly HashSet<ulong> _usages;

            public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
            {
                _links = links;
                _usages = usages;
            }

            public bool Collect(ulong link)
            {
                if (_usages.Add(link))
                {
                    _links.Each(link, _constants.Any, Collect);
                    _links.Each(_constants.Any, link, Collect);
                }
                return true;
            }
        }

        private class AllUsagesCollector1
        {
            private readonly ILinks<ulong> _links;
            private readonly HashSet<ulong> _usages;
            private readonly ulong _continue;

            public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
            {
                _links = links;
                _usages = usages;
                _continue = _links.Constants.Continue;
            }
```

```csharp
        public ulong Collect(IList<ulong> link)
        {
            var linkIndex = _links.GetIndex(link);
            if (_usages.Add(linkIndex))
            {
                _links.Each(Collect, _constants.Any, linkIndex);
            }
            return _continue;
        }
    }

    private class AllUsagesCollector2
    {
        private readonly ILinks<ulong> _links;
        private readonly BitString _usages;

        public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
        {
            _links = links;
            _usages = usages;
        }

        public bool Collect(ulong link)
        {
            if (_usages.Add((long)link))
            {
                _links.Each(link, _constants.Any, Collect);
                _links.Each(_constants.Any, link, Collect);
            }
            return true;
        }
    }

    private class AllUsagesIntersectingCollector
    {
        private readonly SynchronizedLinks<ulong> _links;
        private readonly HashSet<ulong> _intersectWith;
        private readonly HashSet<ulong> _usages;
        private readonly HashSet<ulong> _enter;

        public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
        ↪  intersectWith, HashSet<ulong> usages)
        {
            _links = links;
            _intersectWith = intersectWith;
            _usages = usages;
            _enter = new HashSet<ulong>(); // защита от зацикливания
        }

        public bool Collect(ulong link)
        {
            if (_enter.Add(link))
            {
                if (_intersectWith.Contains(link))
                {
                    _usages.Add(link);
                }
                _links.Unsync.Each(link, _constants.Any, Collect);
                _links.Unsync.Each(_constants.Any, link, Collect);
            }
            return true;
        }
    }

    private void CloseInnerConnections(Action<ulong> handler, ulong left, ulong right)
    {
        TryStepLeftUp(handler, left, right);
        TryStepRightUp(handler, right, left);
    }

    private void AllCloseConnections(Action<ulong> handler, ulong left, ulong right)
    {
        // Direct
        if (left == right)
        {
            handler(left);
        }
        var doublet = Links.Unsync.SearchOrDefault(left, right);
        if (doublet != _constants.Null)
```

```csharp
            {
                handler(doublet);
            }
            // Inner
            CloseInnerConnections(handler, left, right);
            // Outer
            StepLeft(handler, left, right);
            StepRight(handler, left, right);
            PartialStepRight(handler, left, right);
            PartialStepLeft(handler, left, right);
        }

        private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
        ↪ HashSet<ulong> previousMatchings, long startAt)
        {
            if (startAt >= sequence.Length) // ?
            {
                return previousMatchings;
            }
            var secondLinkUsages = new HashSet<ulong>();
            AllUsagesCore(sequence[startAt], secondLinkUsages);
            secondLinkUsages.Add(sequence[startAt]);
            var matchings = new HashSet<ulong>();
            //for (var i = 0; i < previousMatchings.Count; i++)
            foreach (var secondLinkUsage in secondLinkUsages)
            {
                foreach (var previousMatching in previousMatchings)
                {
                    //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
                    ↪ secondLinkUsage);
                    StepRight(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
                    TryStepRightUp(matchings.AddAndReturnVoid, secondLinkUsage,
                    ↪ previousMatching);
                    //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
                    ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
                    ↪ желаемым результам.
                    PartialStepRight(matchings.AddAndReturnVoid, previousMatching,
                    ↪ secondLinkUsage);
                }
            }
            if (matchings.Count == 0)
            {
                return matchings;
            }
            return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
        }

        private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
        ↪ links, params ulong[] sequence)
        {
            if (sequence == null)
            {
                return;
            }
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany &&
                ↪ !links.Exists(sequence[i]))
                {
                    throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                    ↪ $"patternSequence[{i}]");
                }
            }
        }

        // Pattern Matching -> Key To Triggers
        public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                patternSequence = Simplify(patternSequence);
                if (patternSequence.Length > 0)
                {
                    EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
                    var uniqueSequenceElements = new HashSet<ulong>();
                    for (var i = 0; i < patternSequence.Length; i++)
                    {
```

```csharp
                        if (patternSequence[i] != _constants.Any && patternSequence[i] !=
                        ↪   ZeroOrMany)
                        {
                            uniqueSequenceElements.Add(patternSequence[i]);
                        }
                    }
                    var results = new HashSet<ulong>();
                    foreach (var uniqueSequenceElement in uniqueSequenceElements)
                    {
                        AllUsagesCore(uniqueSequenceElement, results);
                    }
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new PatternMatcher(this, patternSequence, filteredResults);
                    matcher.AddAllPatternMatchedToResults(results);
                    return filteredResults;
                }
                return new HashSet<ulong>();
            });
        }

        // Найти все возможные связи между указанным списком связей.
        // Находит связи между всеми указанными связями в любом порядке.
        // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
        ↪   несколько раз в последовательности)
        public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureEachLinkExists(linksToConnect);
                    AllUsagesCore(linksToConnect[0], results);
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var next = new HashSet<ulong>();
                        AllUsagesCore(linksToConnect[i], next);
                        results.IntersectWith(next);
                    }
                }
                return results;
            });
        }

        public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureEachLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector(Links.Unsync, results);
                    collector1.Collect(linksToConnect[0]);
                    var next = new HashSet<ulong>();
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var collector = new AllUsagesCollector(Links.Unsync, next);
                        collector.Collect(linksToConnect[i]);
                        results.IntersectWith(next);
                        next.Clear();
                    }
                }
                return results;
            });
        }

        public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureEachLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector(Links, results);
                    collector1.Collect(linksToConnect[0]);
                    //AllUsagesCore(linksToConnect[0], results);
```

```csharp
                        for (var i = 1; i < linksToConnect.Length; i++)
                        {
                            var next = new HashSet<ulong>();
                            var collector = new AllUsagesIntersectingCollector(Links, results, next);
                            collector.Collect(linksToConnect[i]);
                            //AllUsagesCore(linksToConnect[i], next);
                            //results.IntersectWith(next);
                            results = next;
                        }
                    }
                    return results;
                });
        }

        public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                var results = new BitString((long)Links.Unsync.Count() + 1); // new
                ↪   BitArray((int)_links.Total + 1);
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureEachLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector2(Links.Unsync, results);
                    collector1.Collect(linksToConnect[0]);
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var next = new BitString((long)Links.Unsync.Count() + 1); //new
                        ↪   BitArray((int)_links.Total + 1);
                        var collector = new AllUsagesCollector2(Links.Unsync, next);
                        collector.Collect(linksToConnect[i]);
                        results = results.And(next);
                    }
                }
                return results.GetSetUInt64Indices();
            });
        }

        private static ulong[] Simplify(ulong[] sequence)
        {
            // Считаем новый размер последовательности
            long newLength = 0;
            var zeroOrManyStepped = false;
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] == ZeroOrMany)
                {
                    if (zeroOrManyStepped)
                    {
                        continue;
                    }
                    zeroOrManyStepped = true;
                }
                else
                {
                    //if (zeroOrManyStepped) Is it efficient?
                    zeroOrManyStepped = false;
                }
                newLength++;
            }
            // Строим новую последовательность
            zeroOrManyStepped = false;
            var newSequence = new ulong[newLength];
            long j = 0;
            for (var i = 0; i < sequence.Length; i++)
            {
                //var current = zeroOrManyStepped;
                //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
                //if (current && zeroOrManyStepped)
                //    continue;
                //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
                //if (zeroOrManyStepped && newZeroOrManyStepped)
                //    continue;
                //zeroOrManyStepped = newZeroOrManyStepped;
                if (sequence[i] == ZeroOrMany)
                {
                    if (zeroOrManyStepped)
                    {
                        continue;
```

```csharp
                }
                zeroOrManyStepped = true;
            }
            else
            {
                //if (zeroOrManyStepped) Is it efficient?
                zeroOrManyStepped = false;
            }
            newSequence[j++] = sequence[i];
        }
        return newSequence;
    }

    public static void TestSimplify()
    {
        var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
          ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
        var simplifiedSequence = Simplify(sequence);
    }

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

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

    #region From Triplets

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

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

    private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
      ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
    {
        var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
        var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
        if (leftLinkTotalReferers <= rightLinkTotalReferers)
        {
            var nextLeftLink = middleLinks[leftBound];
            var elements = GetRightElements(leftLink, nextLeftLink);
            if (leftBound <= rightBound)
            {
                for (var i = elements.Length - 1; i >= 0; i--)
                {
                    var element = elements[i];
                    if (element != 0)
                    {
                        CollectMatchingSequences(element, leftBound + 1, middleLinks,
                          ↪ rightLink, rightBound, ref results);
                    }
                }
            }
            else
            {
                for (var i = elements.Length - 1; i >= 0; i--)
                {
                    var element = elements[i];
                    if (element != 0)
                    {
                        results.Add(element);
```

```
1506                                }
1507                            }
1508                        }
1509                    }
1510                    else
1511                    {
1512                        var nextRightLink = middleLinks[rightBound];
1513                        var elements = GetLeftElements(rightLink, nextRightLink);
1514                        if (leftBound <= rightBound)
1515                        {
1516                            for (var i = elements.Length - 1; i >= 0; i--)
1517                            {
1518                                var element = elements[i];
1519                                if (element != 0)
1520                                {
1521                                    CollectMatchingSequences(leftLink, leftBound, middleLinks,
                                        ↪  elements[i], rightBound - 1, ref results);
1522                                }
1523                            }
1524                        }
1525                        else
1526                        {
1527                            for (var i = elements.Length - 1; i >= 0; i--)
1528                            {
1529                                var element = elements[i];
1530                                if (element != 0)
1531                                {
1532                                    results.Add(element);
1533                                }
1534                            }
1535                        }
1536                    }
1537                }

1538
1539            public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1540            {
1541                var result = new ulong[5];
1542                TryStepRight(startLink, rightLink, result, 0);
1543                Links.Each(_constants.Any, startLink, couple =>
1544                {
1545                    if (couple != startLink)
1546                    {
1547                        if (TryStepRight(couple, rightLink, result, 2))
1548                        {
1549                            return false;
1550                        }
1551                    }
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
                            ↪  == Net.And &&
1578                        {
1579                            result[offset + 1] = couple;
1580                            if (++added == 2)
1581                            {
```

```
                                return false;
                            }
                        }
                    }
                    return true;
                });
                return added > 0;
            }

        public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
        {
            var result = new ulong[5];
            TryStepLeft(startLink, leftLink, result, 0);
            Links.Each(startLink, _constants.Any, couple =>
            {
                if (couple != startLink)
                {
                    if (TryStepLeft(couple, leftLink, result, 2))
                    {
                        return false;
                    }
                }
                return true;
            });
            if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
            {
                result[4] = leftLink;
            }
            return result;
        }

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

        #endregion

        #region Walkers

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

            #region Pattern Match

            enum PatternBlockType
            {
                Undefined,
                Gap,
                Elements
```

```csharp
            }

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

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

            #endregion

            public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
            ↪  HashSet<LinkIndex> results)
                : base(sequences.Links.Unsync)
            {
                _sequences = sequences;
                _patternSequence = patternSequence;
                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                ↪  _constants.Any && x != ZeroOrMany));
                _results = results;
                _pattern = CreateDetailedPattern();
            }

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

            public bool PatternMatch(LinkIndex sequenceToMatch)
            {
                _patternPosition = 0;
                _sequencePosition = 0;
                foreach (var part in Walk(sequenceToMatch))
                {
                    if (!PatternMatchCore(Links.GetIndex(part)))
                    {
                        break;
                    }
                }
                return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
                ↪  - 1 && _pattern[_patternPosition].Start == 0);
            }

            private List<PatternBlock> CreateDetailedPattern()
            {
                var pattern = new List<PatternBlock>();
                var patternBlock = new PatternBlock();
                for (var i = 0; i < _patternSequence.Length; i++)
                {
                    if (patternBlock.Type == PatternBlockType.Undefined)
                    {
                        if (_patternSequence[i] == _constants.Any)
                        {
                            patternBlock.Type = PatternBlockType.Gap;
                            patternBlock.Start = 1;
                            patternBlock.Stop = 1;
                        }
                        else if (_patternSequence[i] == ZeroOrMany)
                        {
                            patternBlock.Type = PatternBlockType.Gap;
                            patternBlock.Start = 0;
                            patternBlock.Stop = long.MaxValue;
                        }
                        else
                        {
                            patternBlock.Type = PatternBlockType.Elements;
                            patternBlock.Start = i;
                            patternBlock.Stop = i;
                        }
                    }
                    else if (patternBlock.Type == PatternBlockType.Elements)
                    {
                        if (_patternSequence[i] == _constants.Any)
                        {
                            pattern.Add(patternBlock);
                            patternBlock = new PatternBlock
                            {
                                Type = PatternBlockType.Gap,
```

```csharp
                        Start = 1,
                        Stop = 1
                    };
                }
                else if (_patternSequence[i] == ZeroOrMany)
                {
                    pattern.Add(patternBlock);
                    patternBlock = new PatternBlock
                    {
                        Type = PatternBlockType.Gap,
                        Start = 0,
                        Stop = long.MaxValue
                    };
                }
                else
                {
                    patternBlock.Stop = i;
                }
            }
            else // patternBlock.Type == PatternBlockType.Gap
            {
                if (_patternSequence[i] == _constants.Any)
                {
                    patternBlock.Start++;
                    if (patternBlock.Stop < patternBlock.Start)
                    {
                        patternBlock.Stop = patternBlock.Start;
                    }
                }
                else if (_patternSequence[i] == ZeroOrMany)
                {
                    patternBlock.Stop = long.MaxValue;
                }
                else
                {
                    pattern.Add(patternBlock);
                    patternBlock = new PatternBlock
                    {
                        Type = PatternBlockType.Elements,
                        Start = i,
                        Stop = i
                    };
                }
            }
        }
        if (patternBlock.Type != PatternBlockType.Undefined)
        {
            pattern.Add(patternBlock);
        }
        return pattern;
    }

    ///* match: search for regexp anywhere in text */
    //int match(char* regexp, char* text)
    //{
    //    do
    //    {
    //    } while (*text++ != '\0');
    //    return 0;
    //}

    ///* matchhere: search for regexp at beginning of text */
    //int matchhere(char* regexp, char* text)
    //{
    //    if (regexp[0] == '\0')
    //        return 1;
    //    if (regexp[1] == '*')
    //        return matchstar(regexp[0], regexp + 2, text);
    //    if (regexp[0] == '$' && regexp[1] == '\0')
    //        return *text == '\0';
    //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
    //        return matchhere(regexp + 1, text + 1);
    //    return 0;
    //}

    ///* matchstar: search for c*regexp at beginning of text */
    //int matchstar(int c, char* regexp, char* text)
    //{
    //    do
```

```csharp
            //    {    /* a * matches zero or more instances */
            //        if (matchhere(regexp, text))
            //            return 1;
            //    } while (*text != '\0' && (*text++ == c || c == '.'));
            //    return 0;
            //}

            //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
            ↪  long maximumGap)
            //{
            //    mininumGap = 0;
            //    maximumGap = 0;
            //    element = 0;
            //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
            //    {
            //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
            //            mininumGap++;
            //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
            //            maximumGap = long.MaxValue;
            //        else
            //            break;
            //    }

            //    if (maximumGap < mininumGap)
            //        maximumGap = mininumGap;
            //}

        private bool PatternMatchCore(LinkIndex element)
        {
            if (_patternPosition >= _pattern.Count)
            {
                _patternPosition = -2;
                return false;
            }
            var currentPatternBlock = _pattern[_patternPosition];
            if (currentPatternBlock.Type == PatternBlockType.Gap)
            {
                //var currentMatchingBlockLength = (_sequencePosition -
                ↪  _lastMatchedBlockPosition);
                if (_sequencePosition < currentPatternBlock.Start)
                {
                    _sequencePosition++;
                    return true; // Двигаемся дальше
                }
                // Это последний блок
                if (_pattern.Count == _patternPosition + 1)
                {
                    _patternPosition++;
                    _sequencePosition = 0;
                    return false; // Полное соответствие
                }
                else
                {
                    if (_sequencePosition > currentPatternBlock.Stop)
                    {
                        return false; // Соответствие невозможно
                    }
                    var nextPatternBlock = _pattern[_patternPosition + 1];
                    if (_patternSequence[nextPatternBlock.Start] == element)
                    {
                        if (nextPatternBlock.Start < nextPatternBlock.Stop)
                        {
                            _patternPosition++;
                            _sequencePosition = 1;
                        }
                        else
                        {
                            _patternPosition += 2;
                            _sequencePosition = 0;
                        }
                    }
                }
            }
            else // currentPatternBlock.Type == PatternBlockType.Elements
            {
                var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
                if (_patternSequence[patternElementPosition] != element)
                {
                    return false; // Соответствие невозможно
```

```csharp
                        }
                        if (patternElementPosition == currentPatternBlock.Stop)
                        {
                            _patternPosition++;
                            _sequencePosition = 0;
                        }
                        else
                        {
                            _sequencePosition++;
                        }
                    }
                    return true;
                    //if (_patternSequence[_patternPosition] != element)
                    //    return false;
                    //else
                    //{
                    //    _sequencePosition++;
                    //    _patternPosition++;
                    //    return true;
                    //}
                    ////////////
                    //if (_filterPosition == _patternSequence.Length)
                    //{
                    //    _filterPosition = -2; // Длиннее чем нужно
                    //    return false;
                    //}
                    //if (element != _patternSequence[_filterPosition])
                    //{
                    //    _filterPosition = -1;
                    //    return false; // Начинается иначе
                    //}
                    //_filterPosition++;
                    //if (_filterPosition == (_patternSequence.Length - 1))
                    //    return false;
                    //if (_filterPosition >= 0)
                    //{
                    //    if (element == _patternSequence[_filterPosition + 1])
                    //        _filterPosition++;
                    //    else
                    //        return false;
                    //}
                    //if (_filterPosition < 0)
                    //{
                    //    if (element == _patternSequence[0])
                    //        _filterPosition = 0;
                    //}
                }

                public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
                {
                    foreach (var sequenceToMatch in sequencesToMatch)
                    {
                        if (PatternMatch(sequenceToMatch))
                        {
                            _results.Add(sequenceToMatch);
                        }
                    }
                }
            }

        #endregion
    }
}
```

./Sequences/Sequences.Experiments.ReadSequence.cs

```csharp
//#define USEARRAYPOOL
using System;
using System.Runtime.CompilerServices;
#if USEARRAYPOOL
using Platform.Collections;
#endif

namespace Platform.Data.Doublets.Sequences
{
    partial class Sequences
    {
        public ulong[] ReadSequenceCore(ulong sequence, Func<ulong, bool> isElement)
        {
            var links = Links.Unsync;
```

```csharp
            var length = 1;
            var array = new ulong[length];
            array[0] = sequence;

            if (isElement(sequence))
            {
                return array;
            }

            bool hasElements;
            do
            {
                length *= 2;
#if USEARRAYPOOL
                var nextArray = ArrayPool.Allocate<ulong>(length);
#else
                var nextArray = new ulong[length];
#endif
                hasElements = false;
                for (var i = 0; i < array.Length; i++)
                {
                    var candidate = array[i];
                    if (candidate == 0)
                    {
                        continue;
                    }
                    var doubletOffset = i * 2;
                    if (isElement(candidate))
                    {
                        nextArray[doubletOffset] = candidate;
                    }
                    else
                    {
                        var link = links.GetLink(candidate);
                        var linkSource = links.GetSource(link);
                        var linkTarget = links.GetTarget(link);
                        nextArray[doubletOffset] = linkSource;
                        nextArray[doubletOffset + 1] = linkTarget;
                        if (!hasElements)
                        {
                            hasElements = !(isElement(linkSource) && isElement(linkTarget));
                        }
                    }
                }
#if USEARRAYPOOL
                if (array.Length > 1)
                {
                    ArrayPool.Free(array);
                }
#endif
                array = nextArray;
            }
            while (hasElements);
            var filledElementsCount = CountFilledElements(array);
            if (filledElementsCount == array.Length)
            {
                return array;
            }
            else
            {
                return CopyFilledElements(array, filledElementsCount);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static ulong[] CopyFilledElements(ulong[] array, int filledElementsCount)
        {
            var finalArray = new ulong[filledElementsCount];
            for (int i = 0, j = 0; i < array.Length; i++)
            {
                if (array[i] > 0)
                {
                    finalArray[j] = array[i];
                    j++;
                }
            }
#if USEARRAYPOOL
            ArrayPool.Free(array);
#endif
            return finalArray;
```

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

./Sequences/SequencesExtensions.cs

```
 1    using Platform.Data.Sequences;
 2    using System.Collections.Generic;
 3
 4    namespace Platform.Data.Doublets.Sequences
 5    {
 6        public static class SequencesExtensions
 7        {
 8            public static TLink Create<TLink>(this ISequences<TLink> sequences, IList<TLink[]>
   ↪  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    }
```

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

./Sequences/SequencesOptions.cs

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

```csharp
public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }

public MarkedSequenceCreteriaMatcher<TLink> MarkedSequenceMatcher { get; set; }
public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
public SequencesIndexer<TLink> Indexer { get; set; }

// TODO: Реализовать компактификацию при чтении
//public bool EnforceSingleSequenceVersionOnRead { get; set; }
//public bool UseRequestMarker { get; set; }
//public bool StoreRequestResults { get; set; }

public void InitOptions(ISynchronizedLinks<TLink> links)
{
    if (UseSequenceMarker)
    {
        if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
        {
            SequenceMarkerLink = links.CreatePoint();
        }
        else
        {
            if (!links.Exists(SequenceMarkerLink))
            {
                var link = links.CreatePoint();
                if (!_equalityComparer.Equals(link, SequenceMarkerLink))
                {
                    throw new InvalidOperationException("Cannot recreate sequence marker
                        link.");
                }
            }
        }
        if (MarkedSequenceMatcher == null)
        {
            MarkedSequenceMatcher = new MarkedSequenceCreteriaMatcher<TLink>(links,
                SequenceMarkerLink);
        }
    }
    var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
    if (UseCompression)
    {
        if (LinksToSequenceConverter == null)
        {
            ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
            if (UseSequenceMarker)
            {
                totalSequenceSymbolFrequencyCounter = new
                    TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                    MarkedSequenceMatcher);
            }
            else
            {
                totalSequenceSymbolFrequencyCounter = new
                    TotalSequenceSymbolFrequencyCounter<TLink>(links);
            }
            var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                totalSequenceSymbolFrequencyCounter);
            var compressingConverter = new CompressingConverter<TLink>(links,
                balancedVariantConverter, doubletFrequenciesCache);
            LinksToSequenceConverter = compressingConverter;
        }
    }
    else
    {
        if (LinksToSequenceConverter == null)
        {
            LinksToSequenceConverter = balancedVariantConverter;
        }
    }
    if (UseIndex && Indexer == null)
    {
        Indexer = new SequencesIndexer<TLink>(links);
    }
}

public void ValidateOptions()
{
    if (UseGarbageCollection && !UseSequenceMarker)
    {
```

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

./Sequences/UnicodeMap.cs

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

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

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

./Sequences/Walkers/LeftSequenceWalker.cs

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

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

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

```
43                        {
44                            _stack.Push(element);
45                            element = GetNextElementAfterPush(element);
46                        }
47                    }
48                }
49            }
50
51            [MethodImpl(MethodImplOptions.AggressiveInlining)]
52            protected virtual bool IsElement(IList<TLink> elementLink) =>
       ↪    Point<TLink>.IsPartialPointUnchecked(elementLink);
53
54            [MethodImpl(MethodImplOptions.AggressiveInlining)]
55            protected abstract IList<TLink> GetNextElementAfterPop(IList<TLink> element);
56
57            [MethodImpl(MethodImplOptions.AggressiveInlining)]
58            protected abstract IList<TLink> GetNextElementAfterPush(IList<TLink> element);
59
60            [MethodImpl(MethodImplOptions.AggressiveInlining)]
61            protected abstract IEnumerable<IList<TLink>> WalkContents(IList<TLink> element);
62        }
63    }
```

./Stacks/Stack.cs

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

./Stacks/StackExtensions.cs

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

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

./SynchronizedLinks.cs

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

./UInt64Link.cs

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

```csharp
        private const int Length = 3;

        public readonly ulong Index;
        public readonly ulong Source;
        public readonly ulong Target;

        public static readonly UInt64Link Null = new UInt64Link();

        public UInt64Link(params ulong[] values)
        {
            Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
            ↪  _constants.Null;
            Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
            ↪  _constants.Null;
            Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
            ↪  _constants.Null;
        }

        public UInt64Link(IList<ulong> values)
        {
            Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
            ↪  _constants.Null;
            Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
            ↪  _constants.Null;
            Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
            ↪  _constants.Null;
        }

        public UInt64Link(ulong index, ulong source, ulong target)
        {
            Index = index;
            Source = source;
            Target = target;
        }

        public UInt64Link(ulong source, ulong target)
            : this(_constants.Null, source, target)
        {
            Source = source;
            Target = target;
        }

        public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
        ↪  target);

        public override int GetHashCode() => (Index, Source, Target).GetHashCode();

        public bool IsNull() => Index == _constants.Null
                             && Source == _constants.Null
                             && Target == _constants.Null;

        public override bool Equals(object other) => other is UInt64Link &&
        ↪  Equals((UInt64Link)other);

        public bool Equals(UInt64Link other) => Index == other.Index
                                             && Source == other.Source
                                             && Target == other.Target;

        public static string ToString(ulong index, ulong source, ulong target) => $"({index}:
        ↪  {source}->{target})";

        public static string ToString(ulong source, ulong target) => $"({source}->{target})";

        public static implicit operator ulong[](UInt64Link link) => link.ToArray();

        public static implicit operator UInt64Link(ulong[] linkArray) => new
        ↪  UInt64Link(linkArray);

        public ulong[] ToArray()
        {
            var array = new ulong[Length];
            CopyTo(array, 0);
            return array;
        }

        public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
        ↪  : ToString(Index, Source, Target);

        #region IList
```

```csharp
        public ulong this[int index]
        {
            get
            {
                Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
                →  nameof(index));
                if (index == _constants.IndexPart)
                {
                    return Index;
                }
                if (index == _constants.SourcePart)
                {
                    return Source;
                }
                if (index == _constants.TargetPart)
                {
                    return Target;
                }
                throw new NotSupportedException(); // Impossible path due to
                →  Ensure.ArgumentInRange
            }
            set => throw new NotSupportedException();
        }

        public int Count => Length;

        public bool IsReadOnly => true;

        IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

        public IEnumerator<ulong> GetEnumerator()
        {
            yield return Index;
            yield return Source;
            yield return Target;
        }

        public void Add(ulong item) => throw new NotSupportedException();

        public void Clear() => throw new NotSupportedException();

        public bool Contains(ulong item) => IndexOf(item) >= 0;

        public void CopyTo(ulong[] array, int arrayIndex)
        {
            Ensure.Always.ArgumentNotNull(array, nameof(array));
            Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
            →  nameof(arrayIndex));
            if (arrayIndex + Length > array.Length)
            {
                throw new ArgumentException();
            }
            array[arrayIndex++] = Index;
            array[arrayIndex++] = Source;
            array[arrayIndex] = Target;
        }

        public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();

        public int IndexOf(ulong item)
        {
            if (Index == item)
            {
                return _constants.IndexPart;
            }
            if (Source == item)
            {
                return _constants.SourcePart;
            }
            if (Target == item)
            {
                return _constants.TargetPart;
            }

            return -1;
        }

        public void Insert(int index, ulong item) => throw new NotSupportedException();
```

```csharp
            public void RemoveAt(int index) => throw new NotSupportedException();

            #endregion
        }
    }
```

./UInt64LinkExtensions.cs

```csharp
namespace Platform.Data.Doublets
{
    public static class UInt64LinkExtensions
    {
        public static bool IsFullPoint(this UInt64Link link) => Point<ulong>.IsFullPoint(link);
        public static bool IsPartialPoint(this UInt64Link link) =>
        ↪ Point<ulong>.IsPartialPoint(link);
    }
}
```

./UInt64LinksExtensions.cs

```csharp
using System;
using System.Text;
using System.Collections.Generic;
using Platform.Helpers.Singletons;
using Platform.Data.Constants;
using Platform.Data.Exceptions;
using Platform.Data.Doublets.Sequences;

namespace Platform.Data.Doublets
{
    public static class UInt64LinksExtensions
    {
        public static readonly LinksCombinedConstants<bool, ulong, int> Constants =
        ↪ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;

        public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);

        public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
        {
            if (sequence == null)
            {
                return;
            }
            for (var i = 0; i < sequence.Count; i++)
            {
                if (!links.Exists(sequence[i]))
                {
                    throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                    ↪ $"sequence[{i}]");
                }
            }
        }

        public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
        ↪ sequence)
        {
            if (sequence == null)
            {
                return;
            }
            for (var i = 0; i < sequence.Count; i++)
            {
                if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
                {
                    throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                    ↪ $"sequence[{i}]");
                }
            }
        }

        public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
        {
            if (sequence == null)
            {
                return false;
            }
            var constants = links.Constants;
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] == constants.Any)
                {
```

```csharp
                        return true;
                    }
                }
                return false;
            }

        public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
        ↪  Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
        {
            var sb = new StringBuilder();
            var visited = new HashSet<ulong>();
            links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
            ↪  innerSb.Append(link.Index), renderIndex, renderDebug);
            return sb.ToString();
        }

        public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
        ↪  Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
        ↪  bool renderIndex = false, bool renderDebug = false)
        {
            var sb = new StringBuilder();
            var visited = new HashSet<ulong>();
            links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
            ↪  renderDebug);
            return sb.ToString();
        }

        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)
        {
            if (sb == null)
            {
                throw new ArgumentNullException(nameof(sb));
            }
            if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
            ↪  Constants.Itself)
            {
                return;
            }
            if (links.Exists(linkIndex))
            {
                if (visited.Add(linkIndex))
                {
                    sb.Append('(');
                    var link = new UInt64Link(links.GetLink(linkIndex));
                    if (renderIndex)
                    {
                        sb.Append(link.Index);
                        sb.Append(':');
                    }
                    if (link.Source == link.Index)
                    {
                        sb.Append(link.Index);
                    }
                    else
                    {
                        var source = new UInt64Link(links.GetLink(link.Source));
                        if (isElement(source))
                        {
                            appendElement(sb, source);
                        }
                        else
                        {
                            links.AppendStructure(sb, visited, source.Index, isElement,
                            ↪  appendElement, renderIndex);
                        }
                    }
                    sb.Append(' ');
                    if (link.Target == link.Index)
                    {
                        sb.Append(link.Index);
                    }
                    else
                    {
                        var target = new UInt64Link(links.GetLink(link.Target));
                        if (isElement(target))
```

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

./UInt64LinksTransactionsLayer.cs

```csharp
 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
```

```csharp
        ///     {
        ///         get
        ///         {
        ///             return (ulong) mask & TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public UniqueTimestamp Timestamp
        ///     {
        ///         get
        ///         {
        ///             return (UniqueTimestamp)mask & TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public TransactionItemType Type
        ///     {
        ///         get
        ///         {
        ///             // Использовать по одному биту из TransactionId и Timestamp,
        ///             // для значения в 2 бита, которое представляет тип операции
        ///             throw new NotImplementedException();
        ///         }
        ///     }
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public TransitionHeader Header;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
        /// }
        ///
        /// </remarks>
        public struct Transition
        {
            public static readonly long Size = StructureHelpers.SizeOf<Transition>();

            public readonly ulong TransactionId;
            public readonly UInt64Link Before;
            public readonly UInt64Link After;
            public readonly Timestamp Timestamp;

            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
            ↪  transactionId, UInt64Link before, UInt64Link after)
            {
                TransactionId = transactionId;
                Before = before;
                After = after;
                Timestamp = uniqueTimestampFactory.Create();
            }

            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
            ↪  transactionId, UInt64Link before)
                : this(uniqueTimestampFactory, transactionId, before, default)
            {
            }

            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
                : this(uniqueTimestampFactory, transactionId, default, default)
            {
            }

            public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
            ↪  {After}";
        }

        /// <remarks>
        /// Другие варианты реализации транзакций (атомарности):
        ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
        ↪  Target)) и индексов.
        ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
        ↪  потребуется решить вопрос
        ///         со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
        ↪  пересечениями идентификаторов.
        ///
        /// Где хранить промежуточный список транзакций?
```

```csharp
        ///
        /// В оперативной памяти:
        ///   Минусы:
        ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
        ///     так как нужно отдельно выделять память под список трансформаций.
        ///     2. Выделенной оперативной памяти может не хватить, в том случае,
        ///     если транзакция использует слишком много трансформаций.
        ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
        ///         -> Максимальный размер списка трансформаций можно ограничить / задать
        ///     константой.
        ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
        ///     создавая задержку.
        ///
        /// На жёстком диске:
        ///   Минусы:
        ///     1. Длительный отклик, на запись каждой трансформации.
        ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
        ///         -> Это может решаться упаковкой/исключением дублирующих операций.
        ///         -> Также это может решаться тем, что короткие транзакции вообще
        ///             не будут записываться в случае отката.
        ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
        ///     операции (трансформации)
        ///         будут записаны в лог.
        ///
        /// </remarks>
        public class Transaction : DisposableBase
        {
            private readonly Queue<Transition> _transitions;
            private readonly UInt64LinksTransactionsLayer _layer;
            public bool IsCommitted { get; private set; }
            public bool IsReverted { get; private set; }

            public Transaction(UInt64LinksTransactionsLayer layer)
            {
                _layer = layer;
                if (_layer._currentTransactionId != 0)
                {
                    throw new NotSupportedException("Nested transactions not supported.");
                }
                IsCommitted = false;
                IsReverted = false;
                _transitions = new Queue<Transition>();
                SetCurrentTransaction(layer, this);
            }

            public void Commit()
            {
                EnsureTransactionAllowsWriteOperations(this);
                while (_transitions.Count > 0)
                {
                    var transition = _transitions.Dequeue();
                    _layer._transitions.Enqueue(transition);
                }
                _layer._lastCommitedTransactionId = _layer._currentTransactionId;
                IsCommitted = true;
            }

            private void Revert()
            {
                EnsureTransactionAllowsWriteOperations(this);
                var transitionsToRevert = new Transition[_transitions.Count];
                _transitions.CopyTo(transitionsToRevert, 0);
                for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
                {
                    _layer.RevertTransition(transitionsToRevert[i]);
                }
                IsReverted = true;
            }

            public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
                Transaction transaction)
            {
                layer._currentTransactionId = layer._lastCommitedTransactionId + 1;
                layer._currentTransactionTransitions = transaction._transitions;
                layer._currentTransaction = transaction;
            }

            public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
            {
```

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

```csharp
                var createdLinkIndex = Links.Create();
                var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
                CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  default, createdLink));
                return createdLinkIndex;
            }

        public override ulong Update(IList<ulong> parts)
        {
                var beforeLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
                parts[Constants.IndexPart] = Links.Update(parts);
                var afterLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
                CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  beforeLink, afterLink));
                return parts[Constants.IndexPart];
            }

        public override void Delete(ulong link)
        {
                var deletedLink = new UInt64Link(Links.GetLink(link));
                Links.Delete(link);
                CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  deletedLink, default));
            }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
        ↪  _transitions;

        private void CommitTransition(Transition transition)
        {
                if (_currentTransaction != null)
                {
                        Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
                }
                var transitions = GetCurrentTransitions();
                transitions.Enqueue(transition);
            }

        private void RevertTransition(Transition transition)
        {
                if (transition.After.IsNull()) // Revert Deletion with Creation
                {
                        Links.Create();
                }
                else if (transition.Before.IsNull()) // Revert Creation with Deletion
                {
                        Links.Delete(transition.After.Index);
                }
                else // Revert Update
                {
                        Links.Update(new[] { transition.After.Index, transition.Before.Source,
                        ↪  transition.Before.Target });
                }
            }

        private void ResetCurrentTransation()
        {
                _currentTransactionId = 0;
                _currentTransactionTransitions = null;
                _currentTransaction = null;
            }

        private void PushTransitions()
        {
                if (_log == null || _transitions == null)
                {
                        return;
                }
                for (var i = 0; i < _transitions.Count; i++)
                {
                        var transition = _transitions.Dequeue();

                        _log.Write(transition);
                        _lastCommitedTransition = transition;
                }
            }
```

```csharp
        private void TransitionsPusher()
        {
            while (!IsDisposed && _transitionsPusher != null)
            {
                Thread.Sleep(DefaultPushDelay);
                PushTransitions();
            }
        }

        public Transaction BeginTransaction() => new Transaction(this);

        private void DisposeTransitions()
        {
            try
            {
                var pusher = _transitionsPusher;
                if (pusher != null)
                {
                    _transitionsPusher = null;
                    pusher.Wait();
                }
                if (_transitions != null)
                {
                    PushTransitions();
                }
                Disposable.TryDispose(_log);
                FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
            }
            catch
            {
            }
        }

        #region DisposalBase

        protected override void DisposeCore(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                DisposeTransitions();
            }
            base.DisposeCore(manual, wasDisposed);
        }

        #endregion
    }
}
```

Index