./Platform.Data.Doublets/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 < Type<TLink>.BitsLength; i++)
            {
                if (_equalityComparer.Equals(Arithmetic.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
                    Bit.ShiftRight(number, 1);
                if (_equalityComparer.Equals(number, default))
                {
                    break;
                }
            }
            return target;
        }
    }
}
```

./Platform.Data.Doublets/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 IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;

        public LinkToItsFrequencyNumberConveter(
            ILinks<TLink> links,
            IPropertyOperator<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  }
```

./Platform.Data.Doublets/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  }
```

./Platform.Data.Doublets/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              };
```

```csharp
                var unary = _unaryOne;
                var number = Integer<TLink>.One;
                for (var i = 1; i < 64; i++)
                {
                    _unaryToUInt64.Add(unary = Links.GetOrCreate(unary, unary), number =
                    ↪ (Integer<TLink>)((Integer<TLink>)number * 2UL));
                }
            }

        public TLink Convert(TLink unaryNumber)
        {
            if (_equalityComparer.Equals(unaryNumber, default))
            {
                return default;
            }
            if (_equalityComparer.Equals(unaryNumber, _unaryOne))
            {
                return Integer<TLink>.One;
            }
            var source = Links.GetSource(unaryNumber);
            var target = Links.GetTarget(unaryNumber);
            if (_equalityComparer.Equals(source, target))
            {
                return _unaryToUInt64[unaryNumber];
            }
            else
            {
                var result = _unaryToUInt64[source];
                TLink lastValue;
                while (!_unaryToUInt64.TryGetValue(target, out lastValue))
                {
                    source = Links.GetSource(target);
                    result = Arithmetic.Add(result, _unaryToUInt64[source]);
                    target = Links.GetTarget(target);
                }
                result = Arithmetic.Add(result, lastValue);
                return result;
            }
        }
    }
}
```

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

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

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

        private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;

        public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
        ↪ TLink> powerOf2ToUnaryNumberConverter)
            : base(links)
        {
            _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
            for (int i = 0; i < Type<TLink>.BitsLength; i++)
            {
                _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
            }
        }

        public TLink Convert(TLink sourceNumber)
        {
            var source = sourceNumber;
            var target = Links.Constants.Null;
            while (!_equalityComparer.Equals(source, Links.Constants.Null))
            {
                if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
                {
                    source = Links.Constants.Null;
                }
```

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

./Platform.Data.Doublets/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 }
```

./Platform.Data.Doublets/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 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using Platform.Data.Constants;
4
5  namespace Platform.Data.Doublets.Decorators
6  {
7      public abstract class LinksDecoratorBase<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 }
```

./Platform.Data.Doublets/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  }
```

./Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs
```
 1  using System;
 2  using System.Collections.Generic;
 3  using Platform.Disposables;
 4  using Platform.Data.Constants;
 5
 6  namespace Platform.Data.Doublets.Decorators
 7  {
 8      public abstract class LinksDisposableDecoratorBase<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 Dispose(bool manual, bool wasDisposed)
33          {
34              if (!wasDisposed)
35              {
36                  Links.DisposeIfPossible();
37              }
38          }
39      }
40  }
```

./Platform.Data.Doublets/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      }
```

./Platform.Data.Doublets/Decorators/LinksNonExistentReferencesCreator.cs

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

./Platform.Data.Doublets/Decorators/LinksNullToSelfReferenceResolver.cs

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

./Platform.Data.Doublets/Decorators/LinksSelfReferenceResolver.cs

```
1   using System;
2   using System.Collections.Generic;
3
4   namespace Platform.Data.Doublets.Decorators
5   {
6       public class LinksSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
7       {
8           private static readonly EqualityComparer<TLink> _equalityComparer =
9           ↪ EqualityComparer<TLink>.Default;
10
11          public LinksSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
12
13          public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
```

```
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  }
```

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

```
1   using System.Collections.Generic;
2
3   namespace Platform.Data.Doublets.Decorators
4   {
5       public class LinksUniquenessResolver<TLink> : LinksDecoratorBase<TLink>
6       {
7           private static readonly EqualityComparer<TLink> _equalityComparer =
             ↪ 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  }
```

./Platform.Data.Doublets/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    }
```

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

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

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

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

```csharp
            }
            // TODO: Looks like this is a common type of exceptions linked with restrictions
            //  support
            if (restrictions.Count != 3)
            {
                throw new NotSupportedException();
            }
            var updatedLink = restrictions[Constants.IndexPart];
            this.EnsureLinkExists(updatedLink, nameof(Constants.IndexPart));
            var newSource = restrictions[Constants.SourcePart];
            this.EnsureLinkIsItselfOrExists(newSource, nameof(Constants.SourcePart));
            var newTarget = restrictions[Constants.TargetPart];
            this.EnsureLinkIsItselfOrExists(newTarget, nameof(Constants.TargetPart));
            var existedLink = Constants.Null;
            if (newSource != Constants.Itself && newTarget != Constants.Itself)
            {
                existedLink = this.SearchOrDefault(newSource, newTarget);
            }
            if (existedLink == Constants.Null)
            {
                var before = Links.GetLink(updatedLink);
                if (before[Constants.SourcePart] != newSource || before[Constants.TargetPart] !=
                //  newTarget)
                {
                    Links.Update(updatedLink, newSource == Constants.Itself ? updatedLink :
                    //  newSource,
                                             newTarget == Constants.Itself ? updatedLink :
                                             //  newTarget);
                }
                return updatedLink;
            }
            else
            {
                // Replace one link with another (replaced link is deleted, children are updated
                //  or deleted), it is actually merge operation
                return this.Merge(updatedLink, existedLink);
            }
        }

        /// <summary>Удаляет связь с указанным индексом.</summary>
        /// <param name="link">Индекс удаляемой связи.</param>
        public override void Delete(ulong link)
        {
            this.EnsureLinkExists(link);
            Links.Update(link, Constants.Null, Constants.Null);
            var referencesCount = Links.Count(Constants.Any, link);
            if (referencesCount > 0)
            {
                var references = new ulong[referencesCount];
                var referencesFiller = new ArrayFiller<ulong, ulong>(references,
                //  Constants.Continue);
                Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, link);
                //references.Sort(); // TODO: Решить необходимо ли для корректного порядка
                //  отмены операций в транзакциях
                for (var i = (long)referencesCount - 1; i >= 0; i--)
                {
                    if (this.Exists(references[i]))
                    {
                        Delete(references[i]);
                    }
                }
                //else
                // TODO: Определить почему здесь есть связи, которых не существует
            }
            Links.Delete(link);
        }
    }
}
```

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

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using Platform.Collections;
using Platform.Collections.Arrays;
using Platform.Collections.Lists;
using Platform.Data.Constants;
using Platform.Data.Universal;
```

```csharp
using System.Collections.ObjectModel;

namespace Platform.Data.Doublets.Decorators
{
    /// <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))
////                            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
```

```csharp
147          //{
148          //    //List<IList<T>> matchedLinks = null;
149          //    if (matchedHandler != null)
150          //    {
151          //        matchedLinks = new List<IList<T>>();
152          //        Func<T, bool> handler = link =>
153          //        {
154          //            var matchedLink = Memory.GetLinkValue(link);
155          //            var matchDecision = matchedHandler(matchedLink);
156          //            if (Equals(matchDecision, Constants.Break))
157          //                return false;
158          //            if (!Equals(matchDecision, Constants.Skip))
159          //                matchedLinks.Add(matchedLink);
160          //            return true;
161          //        };
162          //        if (!Memory.Each(handler, restriction))
163          //            return Constants.Break;
164          //    }
165          //    if (!matchedLinks.IsNullOrEmpty())
166          //    {
167          //        var totalMatchedLinks = matchedLinks.Count;
168          //        for (var i = 0; i < totalMatchedLinks; i++)
169          //        {
170          //            var matchedLink = matchedLinks[i];
171          //            if (substitutedHandler != null)
172          //            {
173          //                var newValue = new List<T>(); // TODO: Prepare value to update here
174          //                // TODO: Decide is it actually needed to use Before and After
        ↪  substitution handling.
175          //                var substitutedDecision = substitutedHandler(matchedLink,
        ↪  newValue);
176          //                if (Equals(substitutedDecision, Constants.Break))
177          //                    return Constants.Break;
178          //                if (Equals(substitutedDecision, Constants.Continue))
179          //                {
180          //                    // Actual update here
181          //                    Memory.SetLinkValue(newValue);
182          //                }
183          //                if (Equals(substitutedDecision, Constants.Skip))
184          //                {
185          //                    // Cancel the update. TODO: decide use separate Cancel
        ↪  constant or Skip is enough?
186          //                }
187          //            }
188          //        }
189          //    }
190          //}
191          return Constants.Continue;
192      }
193
194      public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
        ↪  matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
        ↪  substitutionHandler)
195      {
196          if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
197          {
198              return Constants.Continue;
199          }
200          else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
        ↪  Check if it is a correct condition
201          {
202              // Or it only applies to trigger without matchHandler.
203              throw new NotImplementedException();
204          }
205          else if (!substitution.IsNullOrEmpty()) // Creation
206          {
207              var before = ArrayPool<TLink>.Empty;
208              // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
        ↪  (пройти мимо) или пустить (взять)?
209              if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
        ↪  Constants.Break))
210              {
211                  return Constants.Break;
212              }
213              var after = (IList<TLink>)substitution.ToArray();
214              if (_equalityComparer.Equals(after[0], default))
215              {
216                  var newLink = Links.Create();
```

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

```
293                     if (matchHandler != null)
294                     {
295                         return substitutionHandler(before, after);
296                     }
297                     return Constants.Continue;
298                 }
299                 else
300                 {
301                     throw new NotSupportedException();
302                 }
303             }
304         }

306         /// <remarks>
307         /// IList[IList[IList[T]]]
308         /// |        |        |        |||
309         /// |        |         ------  ||
310         /// |        |          link   ||
311         /// |         -------------  |
312         /// |              change      |
313         ///   -------------------
314         ///          changes
315         /// </remarks>
316         public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
    ↪  substitution)
317         {
318             var changes = new List<IList<IList<TLink>>>();
319             Trigger(condition, AlwaysContinue, substitution, (before, after) =>
320             {
321                 var change = new[] { before, after };
322                 changes.Add(change);
323                 return Constants.Continue;
324             });
325             return changes;
326         }

328         private TLink AlwaysContinue(IList<TLink> linkToMatch) => Constants.Continue;
329     }
330 }
```

./Platform.Data.Doublets/DoubletComparer.cs

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

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;

14          public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();

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);

19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public int GetHashCode(Doublet<T> obj) => (obj.Source, obj.Target).GetHashCode();
21      }
22  }
```

./Platform.Data.Doublets/Doublet.cs

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

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;

10          public T Source { get; set; }
11          public T Target { get; set; }
```

```csharp
12
            public Doublet(T source, T target)
13
            {
14
                Source = source;
15
                Target = target;
16
            }
17

18
            public override string ToString() => $"{Source}->{Target}";
19

20
            public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
21
                ↪ && _equalityComparer.Equals(Target, other.Target);
22
            public override bool Equals(object obj) => obj is Doublet<T> doublet ?
23
                ↪ base.Equals(doublet) : false;
24

25
            public override int GetHashCode() => (Source, Target).GetHashCode();
26
        }
27
    }
```

./Platform.Data.Doublets/Hybrid.cs

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

```csharp
            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);

            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();
    }
}
```

./Platform.Data.Doublets/ILinks.cs
```csharp
using Platform.Data.Constants;

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

./Platform.Data.Doublets/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.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)
```

```csharp
        {
            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);
                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 =
            ↪  Arithmetic.Decrement(i))
            {
                links.Delete(i);
                if (!equalityComparer.Equals(links.Count(), Arithmetic.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];
```

```csharp
                //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++)
                {
                    var next = path[i];
                    var values = links.GetLink(current);
                    var source = values[constants.SourcePart];
                    var target = values[constants.TargetPart];
                    if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                        next))
                    {
                        //throw new Exception(string.Format("Невозможно выбрать путь, так как и
                            Source и Target совпадают с элементом пути {0}.", next));
                        return false;
                    }
                    if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                        target))
                    {
                        //throw new Exception(string.Format("Невозможно продолжить путь через
                            элемент пути {0}", next));
                        return false;
                    }
                    current = next;
                }
                return true;
            }

        /// <remarks>
        /// Может потребовать дополнительного стека для PathElement's при использовании
            SequenceWalker.
        /// </remarks>
        public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
            path)
        {
            links.EnsureLinkExists(root, "root");
            var currentLink = root;
            for (var i = 0; i < path.Length; i++)
            {
                currentLink = links.GetLink(currentLink)[path[i]];
            }
            return currentLink;
        }

        public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
            links, TLink root, ulong size, ulong index)
        {
            var constants = links.Constants;
            var source = constants.SourcePart;
            var target = constants.TargetPart;
            if (!Numbers.Math.IsPowerOfTwo(size))
            {
                throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
                    than powers of two are not supported.");
            }
            var path = new BitArray(BitConverter.GetBytes(index));
            var length = Bit.GetLowestPosition(size);
            links.EnsureLinkExists(root, "root");
            var currentLink = root;
            for (var i = length - 1; i >= 0; i--)
            {
                currentLink = links.GetLink(currentLink)[path[i] ? target : source];
            }
            return currentLink;
        }

        #endregion

        /// <summary>
        /// Возвращает индекс указанной связи.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Связь представленная списком, состоящим из её адреса и
            содержимого.</param>
```

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

```csharp
240         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
241         [MethodImpl(MethodImplOptions.AggressiveInlining)]
242         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪ Func<TLink, bool> handler)
243         {
244             var constants = links.Constants;
245             return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
                ↪ constants.Break, constants.Any, source, target);
246         }
247
248         /// <summary>
249         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
250         /// </summary>
251         /// <param name="links">Хранилище связей.</param>
252         /// <param name="source">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
            ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
253         /// <param name="target">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
            ↪ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
254         /// <param name="handler">Обработчик каждой подходящей связи.</param>
255         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
256         [MethodImpl(MethodImplOptions.AggressiveInlining)]
257         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪ Func<IList<TLink>, TLink> handler)
258         {
259             var constants = links.Constants;
260             return links.Each(handler, constants.Any, source, target);
261         }
262
263         [MethodImpl(MethodImplOptions.AggressiveInlining)]
264         public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ restrictions)
265         {
266             var constants = links.Constants;
267             int listSize = (Integer<TLink>)links.Count(restrictions);
268             var list = new IList<TLink>[listSize];
269             if (listSize > 0)
270             {
271                 var filler = new ArrayFiller<IList<TLink>, TLink>(list,
                    ↪ links.Constants.Continue);
272                 links.Each(filler.AddAndReturnConstant, restrictions);
273             }
274             return list;
275         }
276
277         /// <summary>
278         /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
            ↪ в 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)]
```

```csharp
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)]
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        {
```

```csharp
                        var max = nonExistentAddresses.Max();
                        // TODO: Эту верхнюю границу нужно разрешить переопределять (проверить
                        //   ↪  применяется ли эта логика)
                        max = System.Math.Min(max, (Integer<TLink>)constants.MaxPossibleIndex);
                        var createdLinks = new List<TLink>();
                        var equalityComparer = EqualityComparer<TLink>.Default;
                        TLink createdLink = creator();
                        while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
                        {
                            createdLinks.Add(createdLink);
                        }
                        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;
```

```csharp
438            }
439
440            /// <param name="links">Хранилище связей.</param>
441            [MethodImpl(MethodImplOptions.AggressiveInlining)]
442            public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
443            {
444                var link = links.Create();
445                return links.Update(link, link, link);
446            }
447
448            /// <param name="links">Хранилище связей.</param>
449            [MethodImpl(MethodImplOptions.AggressiveInlining)]
450            public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪  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                }
```

```csharp
                    return link;
                }

        /// <summary>
        /// Обновляет связь с указанными началом (Source) и концом (Target)
        /// на связь с указанными началом (NewSource) и концом (NewTarget).
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Индекс связи, которая является началом обновляемой
        ↪   связи.</param>
        /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
        /// <param name="newSource">Индекс связи, которая является началом связи, на которую
        ↪   выполняется обновление.</param>
        /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
        ↪   выполняется обновление.</param>
        /// <returns>Индекс обновлённой связи.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
        ↪   TLink target, TLink newSource, TLink newTarget)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            var link = links.SearchOrDefault(source, target);
            if (equalityComparer.Equals(link, default))
            {
                return links.CreateAndUpdate(newSource, newTarget);
            }
            if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
            ↪   target))
            {
                return link;
            }
            return links.Update(link, newSource, newTarget);
        }

        /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
        /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
        ↪   target)
        {
            var link = links.SearchOrDefault(source, target);
            if (!EqualityComparer<TLink>.Default.Equals(link, default))
            {
                links.Delete(link);
                return link;
            }
            return default;
        }

        /// <summary>Удаляет несколько связей.</summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="deletedLinks">Список адресов связей к удалению.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
        {
            for (int i = 0; i < deletedLinks.Count; i++)
            {
                links.Delete(deletedLinks[i]);
            }
        }

        // Replace one link with another (replaced link is deleted, children are updated or
        ↪   deleted)
        public static TLink Merge<TLink>(this ILinks<TLink> links, TLink linkIndex, TLink
        ↪   newLink)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (equalityComparer.Equals(linkIndex, newLink))
            {
                return newLink;
            }
            var constants = links.Constants;
            ulong referencesAsSourceCount = (Integer<TLink>)links.Count(constants.Any,
            ↪   linkIndex, constants.Any);
            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 }
```

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

```
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Incrementers
5  {
6      public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
7      {
8          private static readonly EqualityComparer<TLink> _equalityComparer =
         ↪   EqualityComparer<TLink>.Default;
9
10         private readonly TLink _frequencyMarker;
11         private readonly TLink _unaryOne;
12         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
13
14         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
         ↪   IIncrementer<TLink> unaryNumberIncrementer)
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 }
```

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

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

namespace Platform.Data.Doublets.Incrementers
{
    public class LinkFrequencyIncrementer<TLink> : LinksOperatorBase<TLink>,
        IIncrementer<IList<TLink>>
    {
        private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
        private readonly IIncrementer<TLink> _frequencyIncrementer;

        public LinkFrequencyIncrementer(ILinks<TLink> links, IPropertyOperator<TLink, TLink>
            frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _frequencyIncrementer = frequencyIncrementer;
        }

        /// <remarks>Sequence itseft is not changed, only frequency of its doublets is
            incremented.</remarks>
        public IList<TLink> Increment(IList<TLink> sequence) // TODO: May be move to
            ILinksExtensions or make SequenceDoubletsFrequencyIncrementer
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
            }
            return sequence;
        }

        public void Increment(TLink link)
        {
            var previousFrequency = _frequencyPropertyOperator.Get(link);
            var frequency = _frequencyIncrementer.Increment(previousFrequency);
            _frequencyPropertyOperator.Set(link, frequency);
        }
    }
}
```

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

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

namespace Platform.Data.Doublets.Incrementers
{
    public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;

        private readonly TLink _unaryOne;

        public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
            _unaryOne = unaryOne;

        public TLink Increment(TLink unaryNumber)
        {
            if (_equalityComparer.Equals(unaryNumber, _unaryOne))
            {
                return Links.GetOrCreate(_unaryOne, _unaryOne);
            }
            var source = Links.GetSource(unaryNumber);
            var target = Links.GetTarget(unaryNumber);
            if (_equalityComparer.Equals(source, target))
            {
                return Links.GetOrCreate(unaryNumber, _unaryOne);
            }
            else
            {
                return Links.GetOrCreate(source, Increment(target));
            }
        }
    }
}
```

./Platform.Data.Doublets/ISynchronizedLinks.cs

```csharp
using Platform.Data.Constants;

```

```csharp
  3    namespace Platform.Data.Doublets
  4    {
  5        public interface ISynchronizedLinks<TLink> : ISynchronizedLinks<TLink, ILinks<TLink>,
           ↪  LinksCombinedConstants<TLink, TLink, int>>, ILinks<TLink>
  6        {
  7        }
  8    }
```

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

```csharp
        public override bool Equals(object other) => other is Link<TLink> &&
        ↪  Equals((Link<TLink>)other);

        public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
                                     && _equalityComparer.Equals(Source, other.Source)
                                     && _equalityComparer.Equals(Target, other.Target);

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

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

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

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

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

        #region IList

        public int Count => Length;

        public bool IsReadOnly => true;

        public TLink 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();
        }

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

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

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

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

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

        public void CopyTo(TLink[] 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 InvalidOperationException();
            }
            array[arrayIndex++] = Index;
            array[arrayIndex++] = Source;
            array[arrayIndex] = Target;
        }
```

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

./Platform.Data.Doublets/LinkExtensions.cs

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

./Platform.Data.Doublets/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  }
```

./Platform.Data.Doublets/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>,
       ↪  IPropertiesOperator<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  }
```

./Platform.Data.Doublets/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>,
    ↪   IPropertyOperator<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);
```

```
            }
          }
        }
      }
```

./Platform.Data.Doublets/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.Singletons;
7  using Platform.Collections.Arrays;
8  using Platform.Numbers;
9  using Platform.Unsafe;
10 using Platform.Memory;
11 using Platform.Data.Exceptions;
12 using Platform.Data.Constants;
13 using static Platform.Numbers.Arithmetic;

15 #pragma warning disable 0649
16 #pragma warning disable 169
17 #pragma warning disable 618

19 // ReSharper disable StaticMemberInGenericType
20 // ReSharper disable BuiltInTypeReferenceStyle
21 // ReSharper disable MemberCanBePrivate.Local
22 // ReSharper disable UnusedMember.Local

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;

31         /// <summary>Возвращает размер одной связи в байтах.</summary>
32         public static readonly int LinkSizeInBytes = Structure<Link>.Size;

34         public static readonly int LinkHeaderSizeInBytes = Structure<LinksHeader>.Size;

36         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;

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();

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;

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
117             public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
    ↪   FreeLinksOffset).GetValue<TLink>();
118             [MethodImpl(MethodImplOptions.AggressiveInlining)]
119             public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
    ↪   FirstFreeLinkOffset).GetValue<TLink>();
120             [MethodImpl(MethodImplOptions.AggressiveInlining)]
121             public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
    ↪   FirstAsSourceOffset).GetValue<TLink>();
122             [MethodImpl(MethodImplOptions.AggressiveInlining)]
123             public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
    ↪   FirstAsTargetOffset).GetValue<TLink>();
124             [MethodImpl(MethodImplOptions.AggressiveInlining)]
125             public static TLink GetLastFreeLink(IntPtr pointer) => (pointer +
    ↪   LastFreeLinkOffset).GetValue<TLink>();

126
127             [MethodImpl(MethodImplOptions.AggressiveInlining)]
128             public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) => pointer +
    ↪   FirstAsSourceOffset;
129             [MethodImpl(MethodImplOptions.AggressiveInlining)]
130             public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
    ↪   FirstAsTargetOffset;

131
132             [MethodImpl(MethodImplOptions.AggressiveInlining)]
133             public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
    ↪   AllocatedLinksOffset).SetValue(value);
134             [MethodImpl(MethodImplOptions.AggressiveInlining)]
135             public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
    ↪   ReservedLinksOffset).SetValue(value);
136             [MethodImpl(MethodImplOptions.AggressiveInlining)]
137             public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
    ↪   FreeLinksOffset).SetValue(value);
138             [MethodImpl(MethodImplOptions.AggressiveInlining)]
139             public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
    ↪   FirstFreeLinkOffset).SetValue(value);
140             [MethodImpl(MethodImplOptions.AggressiveInlining)]
141             public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
    ↪   FirstAsSourceOffset).SetValue(value);
142             [MethodImpl(MethodImplOptions.AggressiveInlining)]
143             public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
    ↪   FirstAsTargetOffset).SetValue(value);
144             [MethodImpl(MethodImplOptions.AggressiveInlining)]
145             public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
    ↪   LastFreeLinkOffset).SetValue(value);
146         }

147
148         private readonly long _memoryReservationStep;

149
150         private readonly IResizableDirectMemory _memory;
151         private IntPtr _header;
152         private IntPtr _links;

153
154         private LinksTargetsTreeMethods _targetsTreeMethods;
155         private LinksSourcesTreeMethods _sourcesTreeMethods;

156
157         // 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
158         private UnusedLinksListMethods _unusedLinksListMethods;

159
160         /// <summary>
161         /// Возвращает общее число связей находящихся в хранилище.
162         /// </summary>
163         private TLink Total => Subtract(LinksHeader.GetAllocatedLinks(_header),
    ↪   LinksHeader.GetFreeLinks(_header));

164
165         public LinksCombinedConstants<TLink, TLink, int> Constants { get; }

166
167         public ResizableDirectMemoryLinks(string address)
168             : this(address, DefaultLinksSizeStep)
169         {
170         }

171
172         /// <summary>
173         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
    ↪   минимальным шагом расширения базы данных.
174         /// </summary>
175         /// <param name="address">Полный 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                     }
```

```csharp
                        return Integer<TLink>.Zero;
                    }
                }
            if (restrictions.Count == 3)
            {
                var index = restrictions[Constants.IndexPart];
                var source = restrictions[Constants.SourcePart];
                var target = restrictions[Constants.TargetPart];

                if (_equalityComparer.Equals(index, Constants.Any))
                {
                    if (_equalityComparer.Equals(source, Constants.Any) &&
                    ↪  _equalityComparer.Equals(target, Constants.Any))
                    {
                        return Total;
                    }
                    else if (_equalityComparer.Equals(source, Constants.Any))
                    {
                        return _targetsTreeMethods.CalculateReferences(target);
                    }
                    else if (_equalityComparer.Equals(target, Constants.Any))
                    {
                        return _sourcesTreeMethods.CalculateReferences(source);
                    }
                    else //if(source != Any && target != Any)
                    {
                        // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
                        var link = _sourcesTreeMethods.Search(source, target);
                        return _equalityComparer.Equals(link, Constants.Null) ?
                        ↪  Integer<TLink>.Zero : Integer<TLink>.One;
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return Integer<TLink>.Zero;
                    }
                    if (_equalityComparer.Equals(source, Constants.Any) &&
                    ↪  _equalityComparer.Equals(target, Constants.Any))
                    {
                        return Integer<TLink>.One;
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (!_equalityComparer.Equals(source, Constants.Any) &&
                    ↪  !_equalityComparer.Equals(target, Constants.Any))
                    {
                        if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
                            _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
                        {
                            return Integer<TLink>.One;
                        }
                        return Integer<TLink>.Zero;
                    }
                    var value = default(TLink);
                    if (_equalityComparer.Equals(source, Constants.Any))
                    {
                        value = target;
                    }
                    if (_equalityComparer.Equals(target, Constants.Any))
                    {
                        value = source;
                    }
                    if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
                        _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
                    {
                        return Integer<TLink>.One;
                    }
                    return Integer<TLink>.Zero;
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪  поддерживаются.");
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
```

```
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                        {
```

```csharp
                        return _targetsTreeMethods.EachReference(target, handler);
                    }
                    else if (_equalityComparer.Equals(target, Constants.Any))
                    {
                        return _sourcesTreeMethods.EachReference(source, handler);
                    }
                    else //if(source != Any && target != Any)
                    {
                        var link = _sourcesTreeMethods.Search(source, target);
                        return _equalityComparer.Equals(link, Constants.Null) ?
                            Constants.Continue : handler(GetLinkStruct(link));
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return Constants.Continue;
                    }
                    if (_equalityComparer.Equals(source, Constants.Any) &&
                        _equalityComparer.Equals(target, Constants.Any))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (!_equalityComparer.Equals(source, Constants.Any) &&
                        !_equalityComparer.Equals(target, Constants.Any))
                    {
                        if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
                            _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
                        {
                            return handler(GetLinkStruct(index));
                        }
                        return Constants.Continue;
                    }
                    var value = default(TLink);
                    if (_equalityComparer.Equals(source, Constants.Any))
                    {
                        value = target;
                    }
                    if (_equalityComparer.Equals(target, Constants.Any))
                    {
                        value = source;
                    }
                    if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
                        _equalityComparer.Equals(Link.GetTarget(storedLinkValue), 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 TLink Update(IList<TLink> 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 (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
            {
                _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
                    linkIndex);
            }
            if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
            {
                _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
                    linkIndex);
            }
            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                  {
```

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

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

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

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

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

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

```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))
```

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

```csharp
            }

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

            protected override TLink GetTreeRoot() => (Header +
            → LinksHeader.FirstAsSourceOffset).GetValue<TLink>();

            protected override TLink GetBasePartValue(TLink link) =>
            → (Links.GetElement(LinkSizeInBytes, link) + Link.SourceOffset).GetValue<TLink>();

            /// <summary>
            /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
            → (концом)
            /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
            /// </summary>
            /// <param name="source">Индекс связи, которая является началом на искомой
            → связи.</param>
            /// <param name="target">Индекс связи, которая является концом на искомой
            → связи.</param>
            /// <returns>Индекс искомой связи.</returns>
            public TLink Search(TLink source, TLink target)
            {
                var root = GetTreeRoot();
                while (!EqualToZero(root))
                {
                    var rootSource = (Links.GetElement(LinkSizeInBytes, root) +
                    → Link.SourceOffset).GetValue<TLink>();
                    var rootTarget = (Links.GetElement(LinkSizeInBytes, root) +
                    → Link.TargetOffset).GetValue<TLink>();
                    if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                    → node.Key < root.Key
                    {
                        root = GetLeftOrDefault(root);
                    }
                    else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
                    → // node.Key > root.Key
                    {
                        root = GetRightOrDefault(root);
                    }
                    else // node.Key == root.Key
                    {
                        return root;
                    }
                }
                return GetZero();
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget, TLink
            → secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
            → (IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget, TLink
            → secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
            → (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
        }

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

```csharp
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 Bit.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(Bit.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>)Bit.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 = Bit.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>)Bit.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 = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
            ↪   1);
339                (Links.GetElement(LinkSizeInBytes, node) +
            ↪   Link.SizeAsTargetOffset).SetValue(modified);
340            }

341

342            protected override sbyte GetBalance(TLink node)
343            {
344                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
            ↪   Link.SizeAsTargetOffset).GetValue<TLink>();
345                var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
```

```csharp
                    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 = Bit.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>();
            }
        }
    }
```

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

```csharp
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Disposables;
using Platform.Collections.Arrays;
using Platform.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
{
    using id = UInt64;

    public unsafe partial class UInt64ResizableDirectMemoryLinks : DisposableBase, ILinks<id>
    {
        /// <summary>Возвращает размер одной связи в байтах.</summary>
        /// <remarks>
```

```csharp
        /// Используется только во вне класса, не рекомедуется использовать внутри.
        /// Так как во вне не обязательно будет доступен unsafe C#.
        /// </remarks>
        public static readonly int LinkSizeInBytes = sizeof(Link);

        public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;

        private struct Link
        {
            public id Source;
            public id Target;
            public id LeftAsSource;
            public id RightAsSource;
            public id SizeAsSource;
            public id LeftAsTarget;
            public id RightAsTarget;
            public id SizeAsTarget;
        }

        private struct LinksHeader
        {
            public id AllocatedLinks;
            public id ReservedLinks;
            public id FreeLinks;
            public id FirstFreeLink;
            public id FirstAsSource;
            public id FirstAsTarget;
            public id LastFreeLink;
            public id Reserved8;
        }

        private readonly long _memoryReservationStep;

        private readonly IResizableDirectMemory _memory;
        private LinksHeader* _header;
        private Link* _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 id Total => _header->AllocatedLinks - _header->FreeLinks;

        // TODO: Дать возможность переопределять в конструкторе
        public LinksCombinedConstants<id, id, int> Constants { get; }

        public UInt64ResizableDirectMemoryLinks(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>
        /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
        ///   6айтах.</param>
        public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
            this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
            memoryReservationStep) { }

        public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
            DefaultLinksSizeStep) { }

        public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            memoryReservationStep)
        {
            Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
            _memory = memory;
            _memoryReservationStep = memoryReservationStep;
            if (memory.ReservedCapacity < memoryReservationStep)
            {
                memory.ReservedCapacity = memoryReservationStep;
            }
            SetPointers(_memory);
```

```csharp
            // 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);
                    }
                    else if (target == Constants.Any)
                    {
                        return _sourcesTreeMethods.CalculateReferences(source);
                    }
                    else //if(source != Any && target != Any)
                    {
```

```
174                 // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
175                 var link = _sourcesTreeMethods.Search(source, target);
176                 return link == Constants.Null ? 0UL : 1UL;
177             }
178         }
179         else
180         {
181             if (!Exists(index))
182             {
183                 return 0;
184             }
185             if (source == Constants.Any && target == Constants.Any)
186             {
187                 return 1;
188             }
189             var storedLinkValue = GetLinkUnsafe(index);
190             if (source != Constants.Any && target != Constants.Any)
191             {
192                 if (storedLinkValue->Source == source &&
193                     storedLinkValue->Target == target)
194                 {
195                     return 1;
196                 }
197                 return 0;
198             }
199             var value = default(id);
200             if (source == Constants.Any)
201             {
202                 value = target;
203             }
204             if (target == Constants.Any)
205             {
206                 value = source;
207             }
208             if (storedLinkValue->Source == value ||
209                 storedLinkValue->Target == value)
210             {
211                 return 1;
212             }
213             return 0;
214         }
215     }
216     throw new NotSupportedException("Другие размеры и способы ограничений не
    ↪ поддерживаются.");
217 }
218
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
221 {
222     if (restrictions.Count == 0)
223     {
224         for (id link = 1; link <= _header->AllocatedLinks; link++)
225         {
226             if (Exists(link))
227             {
228                 if (handler(GetLinkStruct(link)) == Constants.Break)
229                 {
230                     return Constants.Break;
231                 }
232             }
233         }
234         return Constants.Continue;
235     }
236     if (restrictions.Count == 1)
237     {
238         var index = restrictions[Constants.IndexPart];
239         if (index == Constants.Any)
240         {
241             return Each(handler, ArrayPool<ulong>.Empty);
242         }
243         if (!Exists(index))
244         {
245             return Constants.Continue;
246         }
247         return handler(GetLinkStruct(index));
248     }
249     if (restrictions.Count == 2)
250     {
251         var index = restrictions[Constants.IndexPart];
```

```csharp
                    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)
                        {
                            if (storedLinkValue->Source == source &&
                                storedLinkValue->Target == target)
                            {
                                return handler(GetLinkStruct(index));
                            }
                            return Constants.Continue;
                        }
```

```csharp
                        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)
        {
            var link = GetLinkUnsafe(linkIndex);
            return new UInt64Link(linkIndex, link->Source, link->Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private Link* GetLinkUnsafe(id linkIndex) => &_links[linkIndex];
```

```csharp
        /// <remarks>
        /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
        ↪ пространство
        /// </remarks>
        public id Create()
        {
            var freeLink = _header->FirstFreeLink;
            if (freeLink != Constants.Null)
            {
                _unusedLinksListMethods.Detach(freeLink);
            }
            else
            {
                if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
                {
                    throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
                }
                if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
                {
                    _memory.ReservedCapacity += _memoryReservationStep;
                    SetPointers(_memory);
                    _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
                }
                _header->AllocatedLinks++;
                _memory.UsedCapacity += sizeof(Link);
                freeLink = _header->AllocatedLinks;
            }
            return freeLink;
        }

        public void Delete(id link)
        {
            if (link < _header->AllocatedLinks)
            {
                _unusedLinksListMethods.AttachAsFirst(link);
            }
            else if (link == _header->AllocatedLinks)
            {
                _header->AllocatedLinks--;
                _memory.UsedCapacity -= sizeof(Link);
                // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                ↪ пока не дойдём до первой существующей связи
                // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
                while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
                {
                    _unusedLinksListMethods.Detach(_header->AllocatedLinks);
                    _header->AllocatedLinks--;
                    _memory.UsedCapacity -= sizeof(Link);
                }
            }
        }

        /// <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(IResizableDirectMemory memory)
        {
            if (memory == null)
            {
                _header = null;
                _links = null;
                _unusedLinksListMethods = null;
                _targetsTreeMethods = null;
                _unusedLinksListMethods = null;
            }
            else
            {
                _header = (LinksHeader*)(void*)memory.Pointer;
                _links = (Link*)(void*)memory.Pointer;
                _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
                _targetsTreeMethods = new LinksTargetsTreeMethods(this);
                _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
            }
```

```
480              }
481
482              [MethodImpl(MethodImplOptions.AggressiveInlining)]
483              private bool Exists(id link) => link >= Constants.MinPossibleIndex && link <=
        ↪   _header->AllocatedLinks && !IsUnusedLink(link);
484
485              [MethodImpl(MethodImplOptions.AggressiveInlining)]
486              private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
                                      || (_links[link].SizeAsSource == Constants.Null &&
                                      ↪   _links[link].Source != Constants.Null);
488
489          #region Disposable
490
491          protected override bool AllowMultipleDisposeCalls => true;
492
493          protected override void Dispose(bool manual, bool wasDisposed)
494          {
495              if (!wasDisposed)
496              {
497                  SetPointers(null);
498                  _memory.DisposeIfPossible();
499              }
500          }
501
502          #endregion
503      }
504  }
```

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

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

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

```csharp
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using System.Text;
5   using Platform.Collections.Methods.Trees;
6   using Platform.Data.Constants;
7
8   namespace Platform.Data.Doublets.ResizableDirectMemory
```

```csharp
{
    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;
                        root = GetLeftOrDefault(root);
                    }
                }
                root = GetTreeRoot();
                var totalLeftIgnore = 0UL;
                while (root != 0)
                {
                    var @base = GetBasePartValue(root);
                    if (@base >= link)
```

```csharp
                {
                    root = GetLeftOrDefault(root);
                }
                else
                {
                    totalLeftIgnore += GetLeftSize(root) + 1;
                    root = GetRightOrDefault(root);
                }
            }
            return total - totalRightIgnore - totalLeftIgnore;
        }

        public ulong EachReference(ulong link, Func<IList<ulong>, ulong> handler)
        {
            var root = GetTreeRoot();
            if (root == 0)
            {
                return _constants.Continue;
            }
            ulong first = 0, current = root;
            while (current != 0)
            {
                var @base = GetBasePartValue(current);
                if (@base >= link)
                {
                    if (@base == link)
                    {
                        first = current;
                    }
                    current = GetLeftOrDefault(current);
                }
                else
                {
                    current = GetRightOrDefault(current);
                }
            }
            if (first != 0)
            {
                current = first;
                while (true)
                {
                    if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
                    {
                        return _constants.Break;
                    }
                    current = GetNext(current);
                    if (current == 0 || GetBasePartValue(current) != link)
                    {
                        break;
                    }
                }
            }
            return _constants.Continue;
        }

        protected override void PrintNodeValue(ulong node, StringBuilder sb)
        {
            sb.Append(' ');
            sb.Append(Links[node].Source);
            sb.Append('-');
            sb.Append('>');
            sb.Append(Links[node].Target);
        }
    }

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

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

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

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

```csharp
protected override ulong GetRightValue(ulong node) => Links[node].RightAsSource;

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

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

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

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

protected override bool GetLeftIsChild(ulong node)
{
    var previousValue = Links[node].SizeAsSource;
    //return (Integer)Math.PartialRead(previousValue, 4, 1);
    return (previousValue & 16) >> 4 == 1UL;
}

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

protected override bool GetRightIsChild(ulong node)
{
    var previousValue = Links[node].SizeAsSource;
    //return (Integer)Math.PartialRead(previousValue, 3, 1);
    return (previousValue & 8) >> 3 == 1UL;
}

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

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

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

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

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

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

                    /// <summary>
                    /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                    ↪  (концом)
                    /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
                    /// </summary>
                    /// <param name="source">Индекс связи, которая является началом на искомой
                    ↪  связи.</param>
                    /// <param name="target">Индекс связи, которая является концом на искомой
                    ↪  связи.</param>
                    /// <returns>Индекс искомой связи.</returns>
                    public ulong Search(ulong source, ulong target)
                    {
                        var root = Header->FirstAsSource;
                        while (root != 0)
                        {
                            var rootSource = Links[root].Source;
                            var rootTarget = Links[root].Target;
                            if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                            ↪  node.Key < root.Key
                            {
                                root = GetLeftOrDefault(root);
                            }
                            else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
                            ↪  // node.Key > root.Key
                            {
                                root = GetRightOrDefault(root);
                            }
                            else // node.Key == root.Key
                            {
                                return root;
                            }
                        }
                        return 0;
                    }

                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    private static bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
                    ↪  ulong secondSource, ulong secondTarget)
                        => firstSource < secondSource || (firstSource == secondSource && firstTarget <
                        ↪  secondTarget);

                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    private static bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
                    ↪  ulong secondSource, ulong secondTarget)
                        => firstSource > secondSource || (firstSource == secondSource && firstTarget >
                        ↪  secondTarget);

                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    protected override void ClearNode(ulong node)
                    {
                        Links[node].LeftAsSource = 0UL;
                        Links[node].RightAsSource = 0UL;
                        Links[node].SizeAsSource = 0UL;
                    }

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

                    [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)]
```

```csharp
            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;

            //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;
```

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

        protected override ulong GetSize(ulong node)
        {
            var previousValue = Links[node].SizeAsTarget;
            //return Math.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 = Math.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)Math.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 = Math.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)Math.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 = Math.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)
        {
            var previousValue = Links[node].SizeAsTarget;
            //var value = Math.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 = Math.PartialWrite(previousValue, packagedValue, 0, 3);
```

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

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

```
1   using System.Collections.Generic;
2
3   namespace Platform.Data.Doublets.Sequences.Converters
4   {
5       public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
6       {
7           public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
8
9           public override TLink Convert(IList<TLink> sequence)
10          {
11              var length = sequence.Count;
12              if (length < 1)
13              {
14                  return default;
15              }
16              if (length == 1)
17              {
18                  return sequence[0];
19              }
20              // Make copy of next layer
21              if (length > 2)
22              {
23                  // TODO: Try to use stackalloc (which at the moment is not working with
                      ↪ generics) but will be possible with Sigil
24                  var halvedSequence = new TLink[(length / 2) + (length % 2)];
25                  HalveSequence(halvedSequence, sequence, length);
26                  sequence = halvedSequence;
27                  length = halvedSequence.Length;
28              }
29              // Keep creating layer after layer
30              while (length > 2)
31              {
32                  HalveSequence(sequence, sequence, length);
33                  length = (length / 2) + (length % 2);
34              }
35              return Links.GetOrCreate(sequence[0], sequence[1]);
36          }
37
38          private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
39          {
40              var loopedLength = length - (length % 2);
41              for (var i = 0; i < loopedLength; i += 2)
42              {
43                  destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
44              }
45              if (length > loopedLength)
46              {
47                  destination[length / 2] = source[length - 1];
48              }
```

```
49              }
50          }
51      }
```

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

```csharp
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Runtime.CompilerServices;
 4  using Platform.Interfaces;
 5  using Platform.Collections;
 6  using Platform.Singletons;
 7  using Platform.Numbers;
 8  using Platform.Data.Constants;
 9  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
10
11  namespace Platform.Data.Doublets.Sequences.Converters
12  {
13      /// <remarks>
14      /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
        ↪   Links на этапе сжатия.
15      ///     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.
16      ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
        ↪   пар, а так же разом выполнить замену.
17      /// </remarks>
18      public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
19      {
20          private static readonly LinksCombinedConstants<bool, TLink, long> _constants =
            ↪   Default<LinksCombinedConstants<bool, TLink, long>>.Instance;
21          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪   EqualityComparer<TLink>.Default;
22          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
23
24          private readonly IConverter<IList<TLink>, TLink> _baseConverter;
25          private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
26          private readonly TLink _minFrequencyToCompress;
27          private readonly bool _doInitialFrequenciesIncrement;
28          private Doublet<TLink> _maxDoublet;
29          private LinkFrequency<TLink> _maxDoubletData;
30
31          private struct HalfDoublet
32          {
33              public TLink Element;
34              public LinkFrequency<TLink> DoubletData;
35
36              public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
37              {
38                  Element = element;
39                  DoubletData = doubletData;
40              }
41
42              public override string ToString() => $"{Element}: ({DoubletData})";
43          }
44
45          public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            ↪   baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
46              : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
47          {
48          }
49
50          public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            ↪   baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
            ↪   doInitialFrequenciesIncrement)
51              : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
                ↪   doInitialFrequenciesIncrement)
52          {
53          }
54
55          public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            ↪   baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
            ↪   minFrequencyToCompress, bool doInitialFrequenciesIncrement)
56              : base(links)
57          {
58              _baseConverter = baseConverter;
59              _doubletFrequenciesCache = doubletFrequenciesCache;
60              if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
61              {
62                  minFrequencyToCompress = Integer<TLink>.One;
63              }
64              _minFrequencyToCompress = minFrequencyToCompress;
65              _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
```

```csharp
                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>
        /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
        /// </remarks>
        private int ReplaceDoublets(HalfDoublet[] copy)
        {
            var oldLength = copy.Length;
            var newLength = copy.Length;
            while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
            {
                var maxDoubletSource = _maxDoublet.Source;
                var maxDoubletTarget = _maxDoublet.Target;
                if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
                {
                    _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
                }
```

```csharp
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(Arithmetic.Add(doublet.Source, doublet.Target),
                ↪    Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
                ↪    better stability and better compression on sequent data and even on rundom
                ↪    numbers data (but gives collisions anyway) */
210             {
211                 _maxDoublet = doublet;
212                 _maxDoubletData = data;
213             }
214         }
215     }
216 }
```

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

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

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

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

```
47                            var newLevel = i >= length - 1 ?
48                                GetPreviousLowerThanCurrentOrCurrent(previousLevel,
    ↪    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 }
```

./Platform.Data.Doublets/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      }
```

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

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

./Platform.Data.Doublets/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> : ICriterionMatcher<TLink>
7       {
8           private static readonly EqualityComparer<TLink> _equalityComparer =
             ↪  EqualityComparer<TLink>.Default;
9
10          private readonly ILinks<TLink> _links;
11          private readonly TLink _sequenceMarkerLink;
12
13          public 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  }
```

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

```
1   using System.Collections.Generic;
2   using Platform.Collections.Stacks;
3   using Platform.Data.Doublets.Sequences.HeightProviders;
4   using Platform.Data.Sequences;
5
6   namespace Platform.Data.Doublets.Sequences
7   {
8       public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
         ↪  ISequenceAppender<TLink>
9       {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
             ↪  EqualityComparer<TLink>.Default;
11
12          private readonly IStack<TLink> _stack;
13          private readonly ISequenceHeightProvider<TLink> _heightProvider;
14
15          public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
             ↪  ISequenceHeightProvider<TLink> heightProvider)
16              : base(links)
```

```
17          {
18              _stack = stack;
19              _heightProvider = heightProvider;
20          }
21
22          public TLink Append(TLink sequence, TLink appendant)
23          {
24              var cursor = sequence;
25              while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
26              {
27                  var source = Links.GetSource(cursor);
28                  var target = Links.GetTarget(cursor);
29                  if (_equalityComparer.Equals(_heightProvider.Get(source),
                    ↪  _heightProvider.Get(target)))
30                  {
31                      break;
32                  }
33                  else
34                  {
35                      _stack.Push(source);
36                      cursor = target;
37                  }
38              }
39              var left = cursor;
40              var right = appendant;
41              while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
42              {
43                  right = Links.GetOrCreate(left, right);
44                  left = cursor;
45              }
46              return Links.GetOrCreate(left, right);
47          }
48      }
49  }
```

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

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

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

```
1   using System;
2   using System.Linq;
3   using System.Collections.Generic;
4   using Platform.Interfaces;
5   using Platform.Collections;
6   using Platform.Collections.Lists;
7   using Platform.Collections.Segments;
8   using Platform.Collections.Segments.Walkers;
9   using Platform.Singletons;
10  using Platform.Numbers;
11  using Platform.Data.Sequences;
12
13  namespace Platform.Data.Doublets.Sequences
14  {
15      public class DuplicateSegmentsProvider<TLink> :
            ↪  DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
            ↪  IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
16      {
17          private readonly ILinks<TLink> _links;
18          private readonly ISequences<TLink> _sequences;
19          private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
20          private BitString _visited;
21
22          private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
                ↪  IList<TLink>>>
23          {
```

```csharp
                private readonly IListEqualityComparer<TLink> _listComparer;
                public ItemEquilityComparer() => _listComparer =
                →  Default<IListEqualityComparer<TLink>>.Instance;
                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);
                public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
                →  (_listComparer.GetHashCode(pair.Key),
                →  _listComparer.GetHashCode(pair.Value)).GetHashCode();
            }

            private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
            {
                private readonly IListComparer<TLink> _listComparer;

                public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;

                public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
                →  KeyValuePair<IList<TLink>, IList<TLink>> right)
                {
                    var intermediateResult = _listComparer.Compare(left.Key, right.Key);
                    if (intermediateResult == 0)
                    {
                        intermediateResult = _listComparer.Compare(left.Value, right.Value);
                    }
                    return intermediateResult;
                }
            }

            public DuplicateSegmentsProvider(ILinks<TLink> links, ISequences<TLink> sequences)
                : base(minimumStringSegmentLength: 2)
            {
                _links = links;
                _sequences = sequences;
            }

            public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
            {
                _groups = new HashSet<KeyValuePair<IList<TLink>,
                →  IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
                var count = _links.Count();
                _visited = new BitString((long)(Integer<TLink>)count + 1);
                _links.Each(link =>
                {
                    var linkIndex = _links.GetIndex(link);
                    var linkBitIndex = (long)(Integer<TLink>)linkIndex;
                    if (!_visited.Get(linkBitIndex))
                    {
                        var sequenceElements = new List<TLink>();
                        _sequences.EachPart(sequenceElements.AddAndReturnTrue, linkIndex);
                        if (sequenceElements.Count > 2)
                        {
                            WalkAll(sequenceElements);
                        }
                    }
                    return _links.Constants.Continue;
                });
                var resultList = _groups.ToList();
                var comparer = Default<ItemComparer>.Instance;
                resultList.Sort(comparer);
#if DEBUG
                foreach (var item in resultList)
                {
                    PrintDuplicates(item);
                }
#endif
                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)
                {
```

```csharp
                    _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();
        }
    }
}
```

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    public class FrequenciesCacheBasedLinkFrequencyIncrementer<TLink> :
        ↪ IIncrementer<IList<TLink>>
    {
        private readonly LinkFrequenciesCache<TLink> _cache;

```

```
10        public FrequenciesCacheBasedLinkFrequencyIncrementer(LinkFrequenciesCache<TLink> cache)
   ↪  => _cache = cache;
11
12        /// <remarks>Sequence itseft is not changed, only frequency of its doublets is
   ↪  incremented.</remarks>
13        public IList<TLink> Increment(IList<TLink> sequence)
14        {
15            _cache.IncrementFrequencies(sequence);
16            return sequence;
17        }
18    }
19  }
```

./Platform.Data.Doublets/Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToItsFrequencyNumberConver

```
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  }
```

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

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

```csharp
                }
            }

            [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 = Arithmetic.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(Arithmetic.Subtract(frequency, count),
                        ↪  Integer<TLink>.One) > 0))
                         || ((_comparer.Compare(count, frequency) > 0) &&
                            ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency),
                            ↪  Integer<TLink>.One) > 0)))
                        {
                            throw new InvalidOperationException("Frequencies validation failed.");
                        }
                    }
                    //else
                    //{
                    //    if (value.Frequency > 0)
                    //    {
                    //        var frequency = value.Frequency;
                    //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
                    //        var count = _countLinkFrequency(linkIndex);

                    //        if ((frequency > count && frequency - count > 1) || (count > frequency
                    ↪  && count - frequency > 1))
                    //            throw new Exception("Frequencies validation failed.");
                    //    }
```

```
119            //}
120        }
121      }
122    }
123  }
```

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

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

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

```csharp
1   using Platform.Interfaces;
2
3   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
4   {
5       public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
6           SequenceSymbolFrequencyOneOffCounter<TLink>
7       {
7           private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
8
9           public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
10              ICriterionMatcher<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  }
```

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

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

```csharp
        public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
          ↪ TLink symbol)
        {
            _links = links;
            _sequenceLink = sequenceLink;
            _symbol = symbol;
            _total = default;
        }

        public virtual TLink Count()
        {
            if (_comparer.Compare(_total, default) > 0)
            {
                return _total;
            }
            StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
              ↪ IsElement, VisitElement);
            return _total;
        }

        private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
          ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
          ↪ IsPartialPoint

        private bool VisitElement(TLink element)
        {
            if (_equalityComparer.Equals(element, _symbol))
            {
                _total = Arithmetic.Increment(_total);
            }
            return true;
        }
    }
}
```

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

```csharp
using Platform.Interfaces;

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
    {
        private readonly ILinks<TLink> _links;
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

        public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
          ↪ ICriterionMatcher<TLink> markedSequenceMatcher)
        {
            _links = links;
            _markedSequenceMatcher = markedSequenceMatcher;
        }

        public TLink Count(TLink argument) => new
          ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
          ↪ _markedSequenceMatcher, argument).Count();
    }
}
```

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

```csharp
using Platform.Interfaces;
using Platform.Numbers;

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
      ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
    {
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

        public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
          ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol) : base(links, symbol)
            => _markedSequenceMatcher = markedSequenceMatcher;

        protected override void CountSequenceSymbolFrequency(TLink link)
        {
            var symbolFrequencyCounter = new
              ↪ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
              ↪ _markedSequenceMatcher, link, _symbol);
            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
```

```
17              }
18          }
19      }
```

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

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

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

```csharp
1   using System.Collections.Generic;
2   using Platform.Interfaces;
3   using Platform.Numbers;
4
5   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6   {
7       public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
8       {
9           private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
10          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11
12          protected readonly ILinks<TLink> _links;
13          protected readonly TLink _symbol;
14          protected readonly HashSet<TLink> _visits;
15          protected TLink _total;
16
17          public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
18          {
19              _links = links;
20              _symbol = symbol;
21              _visits = new HashSet<TLink>();
22              _total = default;
23          }
24
25          public TLink Count()
26          {
27              if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
28              {
29                  return _total;
30              }
31              CountCore(_symbol);
32              return _total;
33          }
34
35          private void CountCore(TLink link)
36          {
37              var any = _links.Constants.Any;
38              if (_equalityComparer.Equals(_links.Count(any, link), default))
39              {
40                  CountSequenceSymbolFrequency(link);
41              }
42              else
43              {
44                  _links.Each(EachElementHandler, any, link);
45              }
46          }
47
48          protected virtual void CountSequenceSymbolFrequency(TLink link)
49          {
50              var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
                ↪  link, _symbol);
51              _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
52          }
53
54          private TLink EachElementHandler(IList<TLink> doublet)
55          {
56              var constants = _links.Constants;
57              var doubletIndex = doublet[constants.IndexPart];
58              if (_visits.Add(doubletIndex))
59              {
```

```
60              CountCore(doubletIndex);
61          }
62          return constants.Continue;
63      }
64  }
65 }
```

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

```csharp
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 IPropertiesOperator<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             IPropertiesOperator<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 }
```

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

```csharp
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 ICriterionMatcher<TLink> _elementMatcher;
9
10         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<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 = Arithmetic.Increment(height);
20        }
21        return height;
22    }
23 }
24 }
```

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

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

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections;
6  using Platform.Collections.Lists;
7  using Platform.Threading.Synchronization;
8  using Platform.Singletons;
9  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     ///
44     /// Можно ли как-то сделать один общий интерфейс
45     ///
46     ///
47     /// Блокчейн и/или гит для распределённой записи транзакций.
48     ///
49     /// </remarks>
50     public partial class Sequences : ISequences<ulong> // IList<string>, IList<ulong[]> (после
        ↪  завершения реализации Sequences)
51     {
52         private static readonly LinksCombinedConstants<bool, ulong, long> _constants =
            ↪  Default<LinksCombinedConstants<bool, ulong, long>>.Instance;
53
```

```csharp
        /// <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)
            {
                return Links.Count(_constants.Any, Options.SequenceMarkerLink, _constants.Any);
            }
            if (sequence.Length == 1) // Первая связь это адрес
            {
                if (sequence[0] == _constants.Null)
                {
                    return 0;
                }
                if (sequence[0] == _constants.Any)
                {
                    return Count();
                }
            }
```

```csharp
                    if (Options.UseSequenceMarker)
                    {
                        return Links.Count(_constants.Any, Options.SequenceMarkerLink, sequence[0]);
                    }
                    return Links.Exists(sequence[0]) ? 1UL : 0;
                }
                throw new NotImplementedException();
            }

        private ulong CountReferences(params ulong[] restrictions)
        {
            if (restrictions.Length == 0)
            {
                return 0;
            }
            if (restrictions.Length == 1) // Первая связь это адрес
            {
                if (restrictions[0] == _constants.Null)
                {
                    return 0;
                }
                if (Options.UseSequenceMarker)
                {
                    var elementsLink = GetSequenceElements(restrictions[0]);
                    var sequenceLink = GetSequenceByElements(elementsLink);
                    if (sequenceLink != _constants.Null)
                    {
                        return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
                    }
                    return Links.Count(elementsLink);
                }
                return Links.Count(restrictions[0]);
            }
            throw new NotImplementedException();
        }

        #endregion

        #region Create

        public ulong Create(params ulong[] sequence)
        {
            return Sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return _constants.Null;
                }
                Links.EnsureEachLinkExists(sequence);
                return CreateCore(sequence);
            });
        }

        private ulong CreateCore(params ulong[] sequence)
        {
            if (Options.UseIndex)
            {
                Options.Indexer.Index(sequence);
            }
            var sequenceRoot = default(ulong);
            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
            {
                var matches = Each(sequence);
                if (matches.Count > 0)
                {
                    sequenceRoot = matches[0];
                }
            }
            else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
            {
                return CompactCore(sequence);
            }
            if (sequenceRoot == default)
            {
                sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
            }
            if (Options.UseSequenceMarker)
            {
```

```csharp
                        Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
                    }
                    return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
                }

        #endregion

        #region Each

        public List<ulong> Each(params ulong[] sequence)
        {
            var results = new List<ulong>();
            Each(results.AddAndReturnTrue, sequence);
            return results;
        }

        public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return true;
                }
                Links.EnsureEachLinkIsAnyOrExists(sequence);
                if (sequence.Count == 1)
                {
                    var link = sequence[0];
                    if (link == _constants.Any)
                    {
                        return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
                    }
                    return handler(link);
                }
                if (sequence.Count == 2)
                {
                    return Links.Unsync.Each(sequence[0], sequence[1], handler);
                }
                if (Options.UseIndex && !Options.Indexer.CheckIndex(sequence))
                {
                    return false;
                }
                return EachCore(handler, sequence);
            });
        }

        private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
        {
            var matcher = new Matcher(this, sequence, new HashSet<LinkIndex>(), handler);
            // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
            //       Id.
            Func<ulong, bool> innerHandler = Options.UseSequenceMarker ? (Func<ulong,
                bool>)matcher.HandleFullMatchedSequence : matcher.HandleFullMatched;
            //if (sequence.Length >= 2)
            if (!StepRight(innerHandler, sequence[0], sequence[1]))
            {
                return false;
            }
            var last = sequence.Count - 2;
            for (var i = 1; i < last; i++)
            {
                if (!PartialStepRight(innerHandler, sequence[i], sequence[i + 1]))
                {
                    return false;
                }
            }
            if (sequence.Count >= 3)
            {
                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)
        {
```

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

```
362              }

363
364          private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
365          {
366              ulong bestVariant;
367              if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
                  ↪   !sequence.EqualTo(newSequence))
368              {
369                  bestVariant = CompactCore(newSequence);
370              }
371              else
372              {
373                  bestVariant = CreateCore(newSequence);
374              }
375              // TODO: Check all options only ones before loop execution
376              // Возможно нужно две версии Each, возвращающий фактические последовательности и с
                  ↪   маркером,
377              // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
                  ↪   можно получить имея только фактические последовательности.
378              foreach (var variant in Each(sequence))
379              {
380                  if (variant != bestVariant)
381                  {
382                      UpdateOneCore(variant, bestVariant);
383                  }
384              }
385              return bestVariant;
386          }

387
388          private void UpdateOneCore(ulong sequence, ulong newSequence)
389          {
390              if (Options.UseGarbageCollection)
391              {
392                  var sequenceElements = GetSequenceElements(sequence);
393                  var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
394                  var sequenceLink = GetSequenceByElements(sequenceElements);
395                  var newSequenceElements = GetSequenceElements(newSequence);
396                  var newSequenceLink = GetSequenceByElements(newSequenceElements);
397                  if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
398                  {
399                      if (sequenceLink != _constants.Null)
400                      {
401                          Links.Unsync.Merge(sequenceLink, newSequenceLink);
402                      }
403                      Links.Unsync.Merge(sequenceElements, newSequenceElements);
404                  }
405                  ClearGarbage(sequenceElementsContents.Source);
406                  ClearGarbage(sequenceElementsContents.Target);
407              }
408              else
409              {
410                  if (Options.UseSequenceMarker)
411                  {
412                      var sequenceElements = GetSequenceElements(sequence);
413                      var sequenceLink = GetSequenceByElements(sequenceElements);
414                      var newSequenceElements = GetSequenceElements(newSequence);
415                      var newSequenceLink = GetSequenceByElements(newSequenceElements);
416                      if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
417                      {
418                          if (sequenceLink != _constants.Null)
419                          {
420                              Links.Unsync.Merge(sequenceLink, newSequenceLink);
421                          }
422                          Links.Unsync.Merge(sequenceElements, newSequenceElements);
423                      }
424                  }
425                  else
426                  {
427                      if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
428                      {
429                          Links.Unsync.Merge(sequence, newSequence);
430                      }
431                  }
432              }
433          }

434
435          #endregion
436
```

```csharp
        #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>
        public ulong Compact(params ulong[] sequence)
        {
            return Sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return _constants.Null;
                }
                Links.EnsureEachLinkExists(sequence);
                return CompactCore(sequence);
```

```csharp
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private ulong CompactCore(params ulong[] sequence) => UpdateCore(sequence, sequence);

        #endregion

        #region Garbage Collection

        /// <remarks>
        /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
        /// определить извне или в унаследованном классе
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsGarbage(ulong link) => link != Options.SequenceMarkerLink &&
            !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;

        private void ClearGarbage(ulong link)
        {
            if (IsGarbage(link))
            {
                var contents = new UInt64Link(Links.GetLink(link));
                Links.Unsync.Delete(link);
                ClearGarbage(contents.Source);
                ClearGarbage(contents.Target);
            }
        }

        #endregion

        #region Walkers

        public bool EachPart(Func<ulong, bool> handler, ulong sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                var links = Links.Unsync;
                var walker = new RightSequenceWalker<ulong>(links);
                foreach (var part in walker.Walk(sequence))
                {
                    if (!handler(links.GetIndex(part)))
                    {
                        return false;
                    }
                }
                return true;
            });
        }

        public class Matcher : RightSequenceWalker<ulong>
        {
            private readonly Sequences _sequences;
            private readonly IList<LinkIndex> _patternSequence;
            private readonly HashSet<LinkIndex> _linksInSequence;
            private readonly HashSet<LinkIndex> _results;
            private readonly Func<ulong, bool> _stopableHandler;
            private readonly HashSet<ulong> _readAsElements;
            private int _filterPosition;

            public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
                HashSet<LinkIndex> results, Func<LinkIndex, bool> stopableHandler,
                HashSet<LinkIndex> readAsElements = null)
                : base(sequences.Links.Unsync)
            {
                _sequences = sequences;
                _patternSequence = patternSequence;
                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    _constants.Any && x != ZeroOrMany));
                _results = results;
                _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)
            {
```

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

            private bool PartialMatchCore(LinkIndex element)
            {
                if (_filterPosition == (_patternSequence.Count - 1))
                {
                    return false; // Нашлось
                }
                if (_filterPosition >= 0)
```

```csharp
                {
                    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
    }
}
```

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

```csharp
using System;
using LinkIndex = System.UInt64;
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;
```

```csharp
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)Numbers.Math.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;
        }

        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)
```

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

```csharp
                            ulong[] innerSequence = null;
                            Links.Unsync.Each(left, right, doublet =>
                            {
                                if (innerSequence == null)
                                {
                                    innerSequence = new ulong[innerSequenceLength];
                                    for (var isi = 0; isi < linkIndex; isi++)
                                    {
                                        innerSequence[isi] = sequence[isi];
                                    }
                                    for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
                                    {
                                        innerSequence[isi] = sequence[isi + 1];
                                    }
                                }
                                innerSequence[linkIndex] = doublet;
                                Each1(handler, innerSequence);
                                return _constants.Continue;
                            });
                        }
                    }
        }

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

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

        private void EachPartCore(Func<ulong, bool> handler, params ulong[] sequence)
        {
            if (sequence.IsNullOrEmpty())
            {
                return;
            }
            Links.EnsureEachLinkIsAnyOrExists(sequence);
            if (sequence.Length == 1)
            {
                var link = sequence[0];
                if (link > 0)
                {
                    handler(link);
                }
                else
                {
                    Links.Each(_constants.Any, _constants.Any, handler);
                }
            }
            else if (sequence.Length == 2)
            {
                //_links.Each(sequence[0], sequence[1], handler);
                //   o_|         x_o ...
                // x_|           |___|
                Links.Each(sequence[1], _constants.Any, doublet =>
                {
                    var match = Links.SearchOrDefault(sequence[0], doublet);
```

```csharp
                    if (match != _constants.Null)
                    {
                        handler(match);
                    }
                    return true;
                });
                // |_x        ... x_o
                // |_o        |___|
                Links.Each(_constants.Any, sequence[0], doublet =>
                {
                    var match = Links.SearchOrDefault(doublet, sequence[1]);
                    if (match != 0)
                    {
                        handler(match);
                    }
                    return true;
                });
                //           ._x o_.
                //           |___|
                PartialStepRight(x => handler(x), sequence[0], sequence[1]);
            }
            else
            {
                // TODO: Implement other variants
                return;
            }
        }

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

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

        private void TryStepRightUp(Action<ulong> 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)
            {
                handler(stepFrom);
            }
        }

        // TODO: Test
        private void PartialStepLeft(Action<ulong> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(right, _constants.Any, doublet =>
            {
                StepLeft(handler, left, doublet);
                if (right != doublet)
                {
                    PartialStepLeft(handler, left, doublet);
                }
                return true;
            });
        }
```

```csharp
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);
                }
                return results;
            }
            var linksInSequence = new HashSet<ulong>(sequence);
            void handler(ulong result)
            {
                var filterPosition = 0;
                StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
                  Links.Unsync.GetTarget,
                    x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                      x =>
```

```csharp
                        {
                            if (filterPosition == sequence.Length)
                            {
                                filterPosition = -2; // Длиннее чем нужно
                                return false;
                            }
                            if (x != sequence[filterPosition])
                            {
                                filterPosition = -1;
                                return false; // Начинается иначе
                            }
                            filterPosition++;

                            return true;
                        });
                    if (filterPosition == sequence.Length)
                    {
                        results.Add(result);
                    }
                }
                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]);
                }
            }
            return results;
        });
    }

    public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
    {
        return Sync.ExecuteReadOperation(() =>
        {
            var results = new HashSet<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);
                    }
                    return results;
                }
                var matcher = new Matcher(this, sequence, results, null);
                if (sequence.Length >= 2)
                {
                    StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
                }
                var last = sequence.Length - 2;
                for (var i = 1; i < last; i++)
                {
                    PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
                        sequence[i + 1]);
                }
                if (sequence.Length >= 3)
                {
                    StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
                        sequence[sequence.Length - 1]);
                }
```

```csharp
                }
                return results;
            });
        }

        public const int MaxSequenceFormatSize = 200;

        public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
            => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);

        public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
            elementToString, bool insertComma, params LinkIndex[] knownElements) =>
            Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
            elementToString, insertComma, knownElements));

        private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
            Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
            LinkIndex[] knownElements)
        {
            var linksInSequence = new HashSet<ulong>(knownElements);
            //var entered = new HashSet<ulong>();
            var sb = new StringBuilder();
            sb.Append('{');
            if (links.Exists(sequenceLink))
            {
                StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
                    x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
                        entered.AddAndReturnVoid, x => { }, entered.DoNotContains
                    {
                        if (insertComma && sb.Length > 1)
                        {
                            sb.Append(',');
                        }
                        //if (entered.Contains(element))
                        //{
                        //    sb.Append('{');
                        //    elementToString(sb, element);
                        //    sb.Append('}');
                        //}
                        //else
                        elementToString(sb, element);
                        if (sb.Length < MaxSequenceFormatSize)
                        {
                            return true;
                        }
                        sb.Append(insertComma ? ", ..." : "...");
                        return false;
                    });
            }
            sb.Append('}');
            return sb.ToString();
        }

        public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
            knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
            knownElements);

        public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
            LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
            Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
            sequenceLink, elementToString, insertComma, knownElements));

        private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
            Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
            LinkIndex[] knownElements)
        {
            var linksInSequence = new HashSet<ulong>(knownElements);
            var entered = new HashSet<ulong>();
            var sb = new StringBuilder();
            sb.Append('{');
            if (links.Exists(sequenceLink))
            {
                StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
                    x => linksInSequence.Contains(x) || links.IsFullPoint(x),
                        entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
                    {
                        if (insertComma && sb.Length > 1)
                        {
```

```
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                                }
604                                return true;
605                            });
606                        if (filterPosition == (sequence.Length - 1))
607                        {
608                            filteredResults.Add(result);
609                        }
610                    }
611                    return filteredResults;
612                }
613                return new List<ulong>();
614            });
```

```csharp
        }

        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++)
        //            //    AllUsagesCore(sequence[i], results);

        //            var filteredResults = new HashSet<ulong>();
        //            var matcher = new Matcher(this, sequence, filteredResults, null);
        //            matcher.AddAllPartialMatchedToResults(firstResults);
        //            return filteredResults;
        //        }

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

```csharp
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;
757                 }
758                 return new HashSet<ulong>();
759             });
760         }
761
762         // Does not work
763         public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
        ↪    params ulong[] sequence)
764         {
765             var visited = new HashSet<ulong>();
766             var results = new HashSet<ulong>();
```

```csharp
                var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
                  ↪  true; }, readAsElements);
                var last = sequence.Length - 1;
                for (var i = 0; i < last; i++)
                {
                    PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
                }
                return results;
            }

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

```csharp
                    //////            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 потребоваться 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)
        {
            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)
        {
```

```csharp
            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);
            calculator.Calculate();
        }

        private class AllUsagesCalculator
        {
            private readonly SynchronizedLinks<ulong> _links;
            private readonly ulong[] _totals;

            public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
            {
                _links = links;
                _totals = totals;
            }
```

```csharp
          public void Calculate() => _links.Each(_constants.Any, _constants.Any,
              CalculateCore);

          private bool CalculateCore(ulong link)
          {
              if (_totals[link] == 0)
              {
                  var total = 1UL;
                  _totals[link] = total;
                  var visitedChildren = new HashSet<ulong>();
                  bool linkCalculator(ulong child)
                  {
                      if (link != child && visitedChildren.Add(child))
                      {
                          total += _totals[child] == 0 ? 1 : _totals[child];
                      }
                      return true;
                  }
                  _links.Unsync.Each(link, _constants.Any, linkCalculator);
                  _links.Unsync.Each(_constants.Any, link, linkCalculator);
                  _totals[link] = total;
              }
              return true;
          }
      }

      private class AllUsagesCalculator2
      {
          private readonly SynchronizedLinks<ulong> _links;
          private readonly ulong[] _totals;

          public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
          {
              _links = links;
              _totals = totals;
          }

          public void Calculate() => _links.Each(_constants.Any, _constants.Any,
              CalculateCore);

          private bool IsElement(ulong link)
          {
              //_linksInSequence.Contains(link) ||
              return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
                  link;
          }

          private bool CalculateCore(ulong link)
          {
              // TODO: Проработать защиту от зацикливания
              // Основано на SequenceWalker.WalkLeft
              Func<ulong, ulong> getSource = _links.Unsync.GetSource;
              Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
              Func<ulong, bool> isElement = IsElement;
              void visitLeaf(ulong parent)
              {
                  if (link != parent)
                  {
                      _totals[parent]++;
                  }
              }
              void visitNode(ulong parent)
              {
                  if (link != parent)
                  {
                      _totals[parent]++;
                  }
              }
              var stack = new Stack();
              var element = link;
              if (isElement(element))
              {
                  visitLeaf(element);
              }
              else
              {
                  while (true)
                  {
```

```csharp
                    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;
        }

        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
```

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

```csharp
        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++)
                    {
                        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>();
```

```csharp
                    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);
                    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;
            });
```

```csharp
        }

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

```csharp
        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);
                        }
                    }
                }
            }
            else
            {
                var nextRightLink = middleLinks[rightBound];
                var elements = GetLeftElements(rightLink, nextRightLink);
                if (leftBound <= rightBound)
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
```

```csharp
                {
                    var element = elements[i];
                    if (element != 0)
                    {
                        CollectMatchingSequences(leftLink, leftBound, middleLinks,
                            ↪  elements[i], rightBound - 1, ref results);
                    }
                }
            }
            else
            {
                for (var i = elements.Length - 1; i >= 0; i--)
                {
                    var element = elements[i];
                    if (element != 0)
                    {
                        results.Add(element);
                    }
                }
            }
        }
    }

    public ulong[] GetRightElements(ulong startLink, ulong rightLink)
    {
        var result = new ulong[5];
        TryStepRight(startLink, rightLink, result, 0);
        Links.Each(_constants.Any, startLink, couple =>
        {
            if (couple != startLink)
            {
                if (TryStepRight(couple, rightLink, result, 2))
                {
                    return false;
                }
            }
            return true;
        });
        if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
        {
            result[4] = startLink;
        }
        return result;
    }

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

    public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
    {
```

```csharp
            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
            }

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

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

```csharp
            #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,
                                Start = 1,
                                Stop = 1
                            };
                        }
                        else if (_patternSequence[i] == ZeroOrMany)
                        {
                            pattern.Add(patternBlock);
                            patternBlock = new PatternBlock
                            {
                                Type = PatternBlockType.Gap,
                                Start = 0,
                                Stop = long.MaxValue
```

```csharp
                            };
                        }
                        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
            //    {    /* 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;
```

```csharp
//      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
        }
        if (patternElementPosition == currentPatternBlock.Stop)
        {
            _patternPosition++;
            _sequencePosition = 0;
        }
        else
        {
            _sequencePosition++;
        }
    }
    return true;
```

```csharp
                //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
    }
}
```

./Platform.Data.Doublets/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;
            var length = 1;
            var array = new ulong[length];
            array[0] = sequence;

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

            bool hasElements;
            do
            {
```

```
27                    length *= 2;
28  #if USEARRAYPOOL
29                    var nextArray = ArrayPool.Allocate<ulong>(length);
30  #else
31                    var nextArray = new ulong[length];
32  #endif
33                    hasElements = false;
34                    for (var i = 0; i < array.Length; i++)
35                    {
36                        var candidate = array[i];
37                        if (candidate == 0)
38                        {
39                            continue;
40                        }
41                        var doubletOffset = i * 2;
42                        if (isElement(candidate))
43                        {
44                            nextArray[doubletOffset] = candidate;
45                        }
46                        else
47                        {
48                            var link = links.GetLink(candidate);
49                            var linkSource = links.GetSource(link);
50                            var linkTarget = links.GetTarget(link);
51                            nextArray[doubletOffset] = linkSource;
52                            nextArray[doubletOffset + 1] = linkTarget;
53                            if (!hasElements)
54                            {
55                                hasElements = !(isElement(linkSource) && isElement(linkTarget));
56                            }
57                        }
58                    }
59  #if USEARRAYPOOL
60                    if (array.Length > 1)
61                    {
62                        ArrayPool.Free(array);
63                    }
64  #endif
65                    array = nextArray;
66                }
67                while (hasElements);
68                var filledElementsCount = CountFilledElements(array);
69                if (filledElementsCount == array.Length)
70                {
71                    return array;
72                }
73                else
74                {
75                    return CopyFilledElements(array, filledElementsCount);
76                }
77            }
78
79            [MethodImpl(MethodImplOptions.AggressiveInlining)]
80            private static ulong[] CopyFilledElements(ulong[] array, int filledElementsCount)
81            {
82                var finalArray = new ulong[filledElementsCount];
83                for (int i = 0, j = 0; i < array.Length; i++)
84                {
85                    if (array[i] > 0)
86                    {
87                        finalArray[j] = array[i];
88                        j++;
89                    }
90                }
91  #if USEARRAYPOOL
92                ArrayPool.Free(array);
93  #endif
94                return finalArray;
95            }
96
97            [MethodImpl(MethodImplOptions.AggressiveInlining)]
98            private static int CountFilledElements(ulong[] array)
99            {
100               var count = 0;
101               for (var i = 0; i < array.Length; i++)
102               {
103                   if (array[i] > 0)
104                   {
105                       count++;
```

```
106            }
107        }
108        return count;
109    }
110  }
111 }
```

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

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

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

```
1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Sequences
4  {
5      public class SequencesIndexer<TLink>
6      {
7          private static readonly EqualityComparer<TLink> _equalityComparer =
             ↪ EqualityComparer<TLink>.Default;
8
9          private readonly ISynchronizedLinks<TLink> _links;
10         private readonly TLink _null;
11
12         public SequencesIndexer(ISynchronizedLinks<TLink> links)
13         {
14             _links = links;
15             _null = _links.Constants.Null;
16         }
17
18         /// <summary>
19         /// Индексирует последовательность глобально, и возвращает значение,
20         /// определяющие была ли запрошенная последовательность проиндексирована ранее.
21         /// </summary>
22         /// <param name="sequence">Последовательность для индексации.</param>
23         /// <returns>
24         /// True если последовательность уже была проиндексирована ранее и
25         /// False если последовательность была проиндексирована только что.
26         /// </returns>
27         public bool Index(TLink[] sequence)
28         {
29             var indexed = true;
30             var i = sequence.Length;
31             while (--i >= 1 && (indexed =
                ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
                ↪ _null))) { }
32             for (; i >= 1; i--)
33             {
34                 _links.GetOrCreate(sequence[i - 1], sequence[i]);
35             }
36             return indexed;
37         }
38
39         public bool BulkIndex(TLink[] sequence)
40         {
41             var indexed = true;
42             var i = sequence.Length;
43             var links = _links.Unsync;
44             _links.SyncRoot.ExecuteReadOperation(() =>
45             {
```

```
46          while (--i >= 1 && (indexed =
       ↪     !_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 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
5  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
6  using Platform.Data.Doublets.Sequences.Converters;
7  using Platform.Data.Doublets.Sequences.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; }
23         public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
24
25         public MarkedSequenceCreteriaMatcher<TLink> MarkedSequenceMatcher { get; set; }
26         public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
27         public SequencesIndexer<TLink> Indexer { get; set; }
28
29         // TODO: Реализовать компактификацию при чтении
30         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
31         //public bool UseRequestMarker { get; set; }
32         //public bool StoreRequestResults { get; set; }
```

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

./Platform.Data.Doublets/Sequences/UnicodeMap.cs
```csharp
1  using System;
2  using System.Collections.Generic;
```

```csharp
using System.Globalization;
using System.Runtime.CompilerServices;
using System.Text;
using Platform.Data.Sequences;

namespace Platform.Data.Doublets.Sequences
{
    public class UnicodeMap
    {
        public static readonly ulong FirstCharLink = 1;
        public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
        public static readonly ulong MapSize = 1 + char.MaxValue;

        private readonly ILinks<ulong> _links;
        private bool _initialized;

        public UnicodeMap(ILinks<ulong> links) => _links = links;

        public static UnicodeMap InitNew(ILinks<ulong> links)
        {
            var map = new UnicodeMap(links);
            map.Init();
            return map;
        }

        public void Init()
        {
            if (_initialized)
            {
                return;
            }
            _initialized = true;
            var firstLink = _links.CreatePoint();
            if (firstLink != FirstCharLink)
            {
                _links.Delete(firstLink);
            }
            else
            {
                for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
                {
                    // From NIL to It (NIL -> Character) transformation meaning, (or infinite
                    //  ↪ amount of NIL characters before actual Character)
                    var createdLink = _links.CreatePoint();
                    _links.Update(createdLink, firstLink, createdLink);
                    if (createdLink != i)
                    {
                        throw new InvalidOperationException("Unable to initialize UTF 16
                            ↪ table.");
                    }
                }
            }
        }

        // 0 - null link
        // 1 - nil character (0 character)
        // ...
        // 65536 (0(1) + 65535 = 65536 possible values)

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong FromCharToLink(char character) => (ulong)character + 1;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static char FromLinkToChar(ulong link) => (char)(link - 1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsCharLink(ulong link) => link <= MapSize;

        public static string FromLinksToString(IList<ulong> linksList)
        {
            var sb = new StringBuilder();
            for (int i = 0; i < linksList.Count; i++)
            {
                sb.Append(FromLinkToChar(linksList[i]));
            }
            return sb.ToString();
        }

        public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
        {
```

```csharp
            var sb = new StringBuilder();
            if (links.Exists(link))
            {
                StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
                    x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                    ↪  element =>
                    {
                        sb.Append(FromLinkToChar(element));
                        return true;
                    });
            }
            return sb.ToString();
        }

        public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
        ↪  chars.Length);

        public static ulong[] FromCharsToLinkArray(char[] chars, int count)
        {
            // char array to ulong array
            var linksSequence = new ulong[count];
            for (var i = 0; i < count; i++)
            {
                linksSequence[i] = FromCharToLink(chars[i]);
            }
            return linksSequence;
        }

        public static ulong[] FromStringToLinkArray(string sequence)
        {
            // char array to ulong array
            var linksSequence = new ulong[sequence.Length];
            for (var i = 0; i < sequence.Length; i++)
            {
                linksSequence[i] = FromCharToLink(sequence[i]);
            }
            return linksSequence;
        }

        public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
        {
            var result = new List<ulong[]>();
            var offset = 0;
            while (offset < sequence.Length)
            {
                var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
                var relativeLength = 1;
                var absoluteLength = offset + relativeLength;
                while (absoluteLength < sequence.Length &&
                        currentCategory ==
                        ↪  CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
                {
                    relativeLength++;
                    absoluteLength++;
                }
                // char array to ulong array
                var innerSequence = new ulong[relativeLength];
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = FromCharToLink(sequence[i]);
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }

        public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
        {
            var result = new List<ulong[]>();
            var offset = 0;
            while (offset < array.Length)
            {
                var relativeLength = 1;
                if (array[offset] <= LastCharLink)
                {
                    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    }
```

./Platform.Data.Doublets/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   }
```

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

```
1    using System.Collections.Generic;
2    using System.Runtime.CompilerServices;
3
4    namespace Platform.Data.Doublets.Sequences.Walkers
5    {
6        public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
7        {
8            public RightSequenceWalker(ILinks<TLink> links) : base(links) { }
9
10           [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
11         protected override IList<TLink> GetNextElementAfterPop(IList<TLink> element) =>
   ↪       Links.GetLink(Links.GetTarget(element));

13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected override IList<TLink> GetNextElementAfterPush(IList<TLink> element) =>
   ↪       Links.GetLink(Links.GetSource(element));

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 }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Data.Sequences;
4
5  namespace Platform.Data.Doublets.Sequences.Walkers
6  {
7      public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
   ↪       ISequenceWalker<TLink>
8      {
9          // TODO: Use IStack 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    }
```

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

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

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

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

./Platform.Data.Doublets/SynchronizedLinks.cs

```
1    using System;
2    using System.Collections.Generic;
3    using Platform.Data.Constants;
```

```
 4    using Platform.Data.Doublets;
 5    using Platform.Threading.Synchronization;
 6
 7    namespace Platform.Data.Doublets
 8    {
 9        /// <remarks>
10        /// TODO: Autogeneration of synchronized wrapper (decorator).
11        /// TODO: Try to unfold code of each method using IL generation for performance improvements.
12        /// TODO: Or even to unfold multiple layers of implementations.
13        /// </remarks>
14        public class SynchronizedLinks<T> : ISynchronizedLinks<T>
15        {
16            public LinksCombinedConstants<T, T, int> Constants { get; }
17            public ISynchronization SyncRoot { get; }
18            public ILinks<T> Sync { get; }
19            public ILinks<T> Unsync { get; }
20
21            public SynchronizedLinks(ILinks<T> links) : this(new ReaderWriterLockSynchronization(),
                ↪  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    }
```

./Platform.Data.Doublets/UInt64Link.cs

```
 1    using System;
 2    using System.Collections;
 3    using System.Collections.Generic;
 4    using Platform.Exceptions;
 5    using Platform.Ranges;
 6    using Platform.Singletons;
 7    using Platform.Collections.Lists;
 8    using Platform.Data.Constants;
 9
10    namespace Platform.Data.Doublets
11    {
12        /// <summary>
13        /// Структура описывающая уникальную связь.
14        /// </summary>
15        public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
16        {
17            private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
                ↪  Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
18
19            private const int Length = 3;
20
21            public readonly ulong Index;
22            public readonly ulong Source;
23            public readonly ulong Target;
24
25            public static readonly UInt64Link Null = new UInt64Link();
26
```

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

        #region IList

        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;
                }
```

```
  94                  if (index == _constants.TargetPart)
  95                  {
  96                      return Target;
  97                  }
  98                  throw new NotSupportedException(); // Impossible path due to
                      ↪ Ensure.ArgumentInRange
  99              }
 100              set => throw new NotSupportedException();
 101          }

 102
 103          public int Count => Length;

 104
 105          public bool IsReadOnly => true;

 106
 107          IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

 108
 109          public IEnumerator<ulong> GetEnumerator()
 110          {
 111              yield return Index;
 112              yield return Source;
 113              yield return Target;
 114          }

 115
 116          public void Add(ulong item) => throw new NotSupportedException();

 117
 118          public void Clear() => throw new NotSupportedException();

 119
 120          public bool Contains(ulong item) => IndexOf(item) >= 0;

 121
 122          public void CopyTo(ulong[] array, int arrayIndex)
 123          {
 124              Ensure.Always.ArgumentNotNull(array, nameof(array));
 125              Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                      ↪ nameof(arrayIndex));
 126              if (arrayIndex + Length > array.Length)
 127              {
 128                  throw new ArgumentException();
 129              }
 130              array[arrayIndex++] = Index;
 131              array[arrayIndex++] = Source;
 132              array[arrayIndex] = Target;
 133          }

 134
 135          public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();

 136
 137          public int IndexOf(ulong item)
 138          {
 139              if (Index == item)
 140              {
 141                  return _constants.IndexPart;
 142              }
 143              if (Source == item)
 144              {
 145                  return _constants.SourcePart;
 146              }
 147              if (Target == item)
 148              {
 149                  return _constants.TargetPart;
 150              }

 151
 152              return -1;
 153          }

 154
 155          public void Insert(int index, ulong item) => throw new NotSupportedException();

 156
 157          public void RemoveAt(int index) => throw new NotSupportedException();

 158
 159          #endregion
 160      }
 161  }
```

./Platform.Data.Doublets/UInt64LinkExtensions.cs

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

```
8    }

./Platform.Data.Doublets/UInt64LinksExtensions.cs
1    using System;
2    using System.Text;
3    using System.Collections.Generic;
4    using Platform.Singletons;
5    using Platform.Data.Constants;
6    using Platform.Data.Exceptions;
7    using Platform.Data.Doublets.Sequences;
8
9    namespace Platform.Data.Doublets
10   {
11       public static class UInt64LinksExtensions
12       {
13           public static readonly LinksCombinedConstants<bool, ulong, int> Constants =
                 Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
14
15           public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
16
17           public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
18           {
19               if (sequence == null)
20               {
21                   return;
22               }
23               for (var i = 0; i < sequence.Count; i++)
24               {
25                   if (!links.Exists(sequence[i]))
26                   {
27                       throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                         $"sequence[{i}]");
28                   }
29               }
30           }
31
32           public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
                 sequence)
33           {
34               if (sequence == null)
35               {
36                   return;
37               }
38               for (var i = 0; i < sequence.Count; i++)
39               {
40                   if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
41                   {
42                       throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                         $"sequence[{i}]");
43                   }
44               }
45           }
46
47           public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
48           {
49               if (sequence == null)
50               {
51                   return false;
52               }
53               var constants = links.Constants;
54               for (var i = 0; i < sequence.Length; i++)
55               {
56                   if (sequence[i] == constants.Any)
57                   {
58                       return true;
59                   }
60               }
61               return false;
62           }
63
64           public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
                 Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
65           {
66               var sb = new StringBuilder();
67               var visited = new HashSet<ulong>();
68               links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
                     innerSb.Append(link.Index), renderIndex, renderDebug);
69               return sb.ToString();
70           }
```

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

./Platform.Data.Doublets/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
47         ///     {
48         ///         get
49         ///         {
50         ///             return (ulong) mask & TransactionIdCombined;
51         ///         }
52         ///     }
53         ///
54         ///     public UniqueTimestamp Timestamp
55         ///     {
56         ///         get
57         ///         {
58         ///             return (UniqueTimestamp)mask & TransactionIdCombined;
59         ///         }
60         ///     }
61         ///
```

```csharp
///         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 = Structure<Transition>.Size;

    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?
///
/// 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:
```

```csharp
        ///     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)
            {
                if (transaction.IsReverted)
                {
                    throw new InvalidOperationException("Transation is reverted.");
                }
                if (transaction.IsCommitted)
                {
                    throw new InvalidOperationException("Transation is commited.");
                }
            }

            protected override void Dispose(bool manual, bool wasDisposed)
            {
                if (!wasDisposed && _layer != null && !_layer.IsDisposed)
                {
                    if (!IsCommitted && !IsReverted)
```

```
209                         {
210                             Revert();
211                         }
212                         _layer.ResetCurrentTransation();
213                     }
214                 }
215             }
216
217         public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
218
219         private readonly string _logAddress;
220         private readonly FileStream _log;
221         private readonly Queue<Transition> _transitions;
222         private readonly UniqueTimestampFactory _uniqueTimestampFactory;
223         private Task _transitionsPusher;
224         private Transition _lastCommitedTransition;
225         private ulong _currentTransactionId;
226         private Queue<Transition> _currentTransactionTransitions;
227         private Transaction _currentTransaction;
228         private ulong _lastCommitedTransactionId;
229
230         public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
231             : base(links)
232         {
233             if (string.IsNullOrWhiteSpace(logAddress))
234             {
235                 throw new ArgumentNullException(nameof(logAddress));
236             }
237             // В первой строке файла хранится последняя закоммиченную транзакцию.
238             // При запуске это используется для проверки удачного закрытия файла лога.
239             // In the first line of the file the last committed transaction is stored.
240             // On startup, this is used to check that the log file is successfully closed.
241             var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
242             var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
243             if (!lastCommitedTransition.Equals(lastWrittenTransition))
244             {
245                 Dispose();
246                 throw new NotSupportedException("Database is damaged, autorecovery is not
                     ↪  supported yet.");
247             }
248             if (lastCommitedTransition.Equals(default(Transition)))
249             {
250                 FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
251             }
252             _lastCommitedTransition = lastCommitedTransition;
253             // TODO: Think about a better way to calculate or store this value
254             var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
255             _lastCommitedTransactionId = allTransitions.Max(x => x.TransactionId);
256             _uniqueTimestampFactory = new UniqueTimestampFactory();
257             _logAddress = logAddress;
258             _log = FileHelpers.Append(logAddress);
259             _transitions = new Queue<Transition>();
260             _transitionsPusher = new Task(TransitionsPusher);
261             _transitionsPusher.Start();
262         }
263
264         public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
265
266         public override ulong Create()
267         {
268             var createdLinkIndex = Links.Create();
269             var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
270             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                 ↪  default, createdLink));
271             return createdLinkIndex;
272         }
273
274         public override ulong Update(IList<ulong> parts)
275         {
276             var beforeLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
277             parts[Constants.IndexPart] = Links.Update(parts);
278             var afterLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
279             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                 ↪  beforeLink, afterLink));
280             return parts[Constants.IndexPart];
281         }
282
283         public override void Delete(ulong link)
284         {
```

```
285        var deletedLink = new UInt64Link(Links.GetLink(link));
286        Links.Delete(link);
287        CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
           ↪   deletedLink, default));
288    }
289
290    [MethodImpl(MethodImplOptions.AggressiveInlining)]
291    private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
       ↪   _transitions;
292
293    private void CommitTransition(Transition transition)
294    {
295        if (_currentTransaction != null)
296        {
297            Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
298        }
299        var transitions = GetCurrentTransitions();
300        transitions.Enqueue(transition);
301    }
302
303    private void RevertTransition(Transition transition)
304    {
305        if (transition.After.IsNull()) // Revert Deletion with Creation
306        {
307            Links.Create();
308        }
309        else if (transition.Before.IsNull()) // Revert Creation with Deletion
310        {
311            Links.Delete(transition.After.Index);
312        }
313        else // Revert Update
314        {
315            Links.Update(new[] { transition.After.Index, transition.Before.Source,
               ↪   transition.Before.Target });
316        }
317    }
318
319    private void ResetCurrentTransation()
320    {
321        _currentTransactionId = 0;
322        _currentTransactionTransitions = null;
323        _currentTransaction = null;
324    }
325
326    private void PushTransitions()
327    {
328        if (_log == null || _transitions == null)
329        {
330            return;
331        }
332        for (var i = 0; i < _transitions.Count; i++)
333        {
334            var transition = _transitions.Dequeue();
335
336            _log.Write(transition);
337            _lastCommitedTransition = transition;
338        }
339    }
340
341    private void TransitionsPusher()
342    {
343        while (!IsDisposed && _transitionsPusher != null)
344        {
345            Thread.Sleep(DefaultPushDelay);
346            PushTransitions();
347        }
348    }
349
350    public Transaction BeginTransaction() => new Transaction(this);
351
352    private void DisposeTransitions()
353    {
354        try
355        {
356            var pusher = _transitionsPusher;
357            if (pusher != null)
358            {
359                _transitionsPusher = null;
360                pusher.Wait();
```

```
361            }
362            if (_transitions != null)
363            {
364                PushTransitions();
365            }
366            _log.DisposeIfPossible();
367            FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
368        }
369        catch
370        {
371        }
372    }
373
374    #region DisposalBase
375
376    protected override void Dispose(bool manual, bool wasDisposed)
377    {
378        if (!wasDisposed)
379        {
380            DisposeTransitions();
381        }
382        base.Dispose(manual, wasDisposed);
383    }
384
385    #endregion
386    }
387 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using Xunit;
4  using Platform.Diagnostics;
5
6  namespace Platform.Tests
7  {
8      public static class ComparisonTests
9      {
10         protected class UInt64Comparer : IComparer<ulong>
11         {
12             public int Compare(ulong x, ulong y) => x.CompareTo(y);
13         }
14
15         private static int Compare(ulong x, ulong y) => x.CompareTo(y);
16
17         [Fact]
18         public static void GreaterOrEqualPerfomanceTest()
19         {
20             const int N = 1000000;
21
22             ulong x = 10;
23             ulong y = 500;
24
25             bool result = false;
26
27             var ts1 = Performance.Measure(() =>
28             {
29                 for (int i = 0; i < N; i++)
30                 {
31                     result = Compare(x, y) >= 0;
32                 }
33             });
34
35             var comparer1 = Comparer<ulong>.Default;
36
37             var ts2 = Performance.Measure(() =>
38             {
39                 for (int i = 0; i < N; i++)
40                 {
41                     result = comparer1.Compare(x, y) >= 0;
42                 }
43             });
44
45             Func<ulong, ulong, int> compareReference = comparer1.Compare;
46
47             var ts3 = Performance.Measure(() =>
48             {
49                 for (int i = 0; i < N; i++)
50                 {
51                     result = compareReference(x, y) >= 0;
```

```
52              }
53          });

55          var comparer2 = new UInt64Comparer();

57          var ts4 = Performance.Measure(() =>
58          {
59              for (int i = 0; i < N; i++)
60              {
61                  result = comparer2.Compare(x, y) >= 0;
62              }
63          });

65          Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");
66      }
67   }
68 }
```

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

```
1  using System.Collections.Generic;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Numbers;
5  using Platform.Memory;
6  using Platform.Scopes;
7  using Platform.Setters;
8  using Platform.Data.Doublets.ResizableDirectMemory;
9
10 namespace Platform.Data.Doublets.Tests
11 {
12     public static class DoubletLinksTests
13     {
14         [Fact]
15         public static void UInt64CRUDTest()
16         {
17             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                 ↪  ResizableDirectMemoryLinks<ulong>>>())
18             {
19                 scope.Use<ILinks<ulong>>().TestCRUDOperations();
20             }
21         }
22
23         [Fact]
24         public static void UInt32CRUDTest()
25         {
26             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                 ↪  ResizableDirectMemoryLinks<uint>>>())
27             {
28                 scope.Use<ILinks<uint>>().TestCRUDOperations();
29             }
30         }
31
32         [Fact]
33         public static void UInt16CRUDTest()
34         {
35             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                 ↪  ResizableDirectMemoryLinks<ushort>>>())
36             {
37                 scope.Use<ILinks<ushort>>().TestCRUDOperations();
38             }
39         }
40
41         [Fact]
42         public static void UInt8CRUDTest()
43         {
44             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                 ↪  ResizableDirectMemoryLinks<byte>>>())
45             {
46                 scope.Use<ILinks<byte>>().TestCRUDOperations();
47             }
48         }
49
50         private static void TestCRUDOperations<T>(this ILinks<T> links)
51         {
52             var constants = links.Constants;
53
54             var equalityComparer = EqualityComparer<T>.Default;
55
56             // Create Link
```

```csharp
                    Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));

                    var setter = new Setter<T>(constants.Null);
                    links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

                    Assert.True(equalityComparer.Equals(setter.Result, constants.Null));

                    var linkAddress = links.Create();

                    var link = new Link<T>(links.GetLink(linkAddress));

                    Assert.True(link.Count == 3);
                    Assert.True(equalityComparer.Equals(link.Index, linkAddress));
                    Assert.True(equalityComparer.Equals(link.Source, constants.Null));
                    Assert.True(equalityComparer.Equals(link.Target, constants.Null));

                    Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.One));

                    // Get first link
                    setter = new Setter<T>(constants.Null);
                    links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);

                    Assert.True(equalityComparer.Equals(setter.Result, linkAddress));

                    // Update link to reference itself
                    links.Update(linkAddress, linkAddress, linkAddress);

                    link = new Link<T>(links.GetLink(linkAddress));

                    Assert.True(equalityComparer.Equals(link.Source, linkAddress));
                    Assert.True(equalityComparer.Equals(link.Target, linkAddress));

                    // Update link to reference null (prepare for delete)
                    var updated = links.Update(linkAddress, constants.Null, constants.Null);

                    Assert.True(equalityComparer.Equals(updated, linkAddress));

                    link = new Link<T>(links.GetLink(linkAddress));

                    Assert.True(equalityComparer.Equals(link.Source, constants.Null));
                    Assert.True(equalityComparer.Equals(link.Target, constants.Null));

                    // Delete link
                    links.Delete(linkAddress);

                    Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));

                    setter = new Setter<T>(constants.Null);
                    links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

                    Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
        }

        [Fact]
        public static void UInt64RawNumbersCRUDTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
              ResizableDirectMemoryLinks<ulong>>>())
            {
                scope.Use<ILinks<ulong>>().TestRawNumbersCRUDOperations();
            }
        }

        [Fact]
        public static void UInt32RawNumbersCRUDTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
              ResizableDirectMemoryLinks<uint>>>())
            {
                scope.Use<ILinks<uint>>().TestRawNumbersCRUDOperations();
            }
        }

        [Fact]
        public static void UInt16RawNumbersCRUDTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
              ResizableDirectMemoryLinks<ushort>>>())
            {
```

```
133                    scope.Use<ILinks<ushort>>().TestRawNumbersCRUDOperations();
134                }
135            }
136
137            [Fact]
138            public static void UInt8RawNumbersCRUDTest()
139            {
140                using (var scope = new Scope<Types<HeapResizableDirectMemory,
                   ↪  ResizableDirectMemoryLinks<byte>>>())
141                {
142                    scope.Use<ILinks<byte>>().TestRawNumbersCRUDOperations();
143                }
144            }
145
146            private static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
147            {
148                // Constants
149                var constants = links.Constants;
150                var equalityComparer = EqualityComparer<T>.Default;
151
152                var h106E = new Hybrid<T>(106L, isExternal: true);
153                var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
154                var h108E = new Hybrid<T>(-108L);
155
156                Assert.Equal(106L, h106E.AbsoluteValue);
157                Assert.Equal(107L, h107E.AbsoluteValue);
158                Assert.Equal(108L, h108E.AbsoluteValue);
159
160                // Create Link (External -> External)
161                var linkAddress1 = links.Create();
162
163                links.Update(linkAddress1, h106E, h108E);
164
165                var link1 = new Link<T>(links.GetLink(linkAddress1));
166
167                Assert.True(equalityComparer.Equals(link1.Source, h106E));
168                Assert.True(equalityComparer.Equals(link1.Target, h108E));
169
170                // Create Link (Internal -> External)
171                var linkAddress2 = links.Create();
172
173                links.Update(linkAddress2, linkAddress1, h108E);
174
175                var link2 = new Link<T>(links.GetLink(linkAddress2));
176
177                Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
178                Assert.True(equalityComparer.Equals(link2.Target, h108E));
179
180                // Create Link (Internal -> Internal)
181                var linkAddress3 = links.Create();
182
183                links.Update(linkAddress3, linkAddress1, linkAddress2);
184
185                var link3 = new Link<T>(links.GetLink(linkAddress3));
186
187                Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
188                Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
189
190                // Search for created link
191                var setter1 = new Setter<T>(constants.Null);
192                links.Each(h106E, h108E, setter1.SetAndReturnFalse);
193
194                Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
195
196                // Search for nonexistent link
197                var setter2 = new Setter<T>(constants.Null);
198                links.Each(h106E, h107E, setter2.SetAndReturnFalse);
199
200                Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
201
202                // Update link to reference null (prepare for delete)
203                var updated = links.Update(linkAddress3, constants.Null, constants.Null);
204
205                Assert.True(equalityComparer.Equals(updated, linkAddress3));
206
207                link3 = new Link<T>(links.GetLink(linkAddress3));
208
209                Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
210                Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
211
```

```
212          // Delete link
213          links.Delete(linkAddress3);
214
215          Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));
216
217          var setter3 = new Setter<T>(constants.Null);
218          links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
219
220          Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
221      }
222
223      // TODO: Test layers
224  }
225 }
```

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

```csharp
1  using System;
2  using System.Collections.Generic;
3  using Xunit;
4  using Platform.Diagnostics;
5
6  namespace Platform.Tests
7  {
8      public static class EqualityTests
9      {
10         protected class UInt64EqualityComparer : IEqualityComparer<ulong>
11         {
12             public bool Equals(ulong x, ulong y) => x == y;
13
14             public int GetHashCode(ulong obj) => obj.GetHashCode();
15         }
16
17         private static bool Equals1<T>(T x, T y) => Equals(x, y);
18
19         private static bool Equals2<T>(T x, T y) => x.Equals(y);
20
21         private static bool Equals3(ulong x, ulong y) => x == y;
22
23         [Fact]
24         public static void EqualsPerfomanceTest()
25         {
26             const int N = 1000000;
27
28             ulong x = 10;
29             ulong y = 500;
30
31             bool result = false;
32
33             var ts1 = Performance.Measure(() =>
34             {
35                 for (int i = 0; i < N; i++)
36                 {
37                     result = Equals1(x, y);
38                 }
39             });
40
41             var ts2 = Performance.Measure(() =>
42             {
43                 for (int i = 0; i < N; i++)
44                 {
45                     result = Equals2(x, y);
46                 }
47             });
48
49             var ts3 = Performance.Measure(() =>
50             {
51                 for (int i = 0; i < N; i++)
52                 {
53                     result = Equals3(x, y);
54                 }
55             });
56
57             var equalityComparer1 = EqualityComparer<ulong>.Default;
58
59             var ts4 = Performance.Measure(() =>
60             {
61                 for (int i = 0; i < N; i++)
62                 {
63                     result = equalityComparer1.Equals(x, y);
64                 }
```

```csharp
                });

                var equalityComparer2 = new UInt64EqualityComparer();

                var ts5 = Performance.Measure(() =>
                {
                    for (int i = 0; i < N; i++)
                    {
                        result = equalityComparer2.Equals(x, y);
                    }
                });

                Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;

                var ts6 = Performance.Measure(() =>
                {
                    for (int i = 0; i < N; i++)
                    {
                        result = equalityComparer3(x, y);
                    }
                });

                var comparer = Comparer<ulong>.Default;

                var ts7 = Performance.Measure(() =>
                {
                    for (int i = 0; i < N; i++)
                    {
                        result = comparer.Compare(x, y) == 0;
                    }
                });

                Assert.True(ts2 < ts1);
                Assert.True(ts3 < ts2);
                Assert.True(ts5 < ts4);
                Assert.True(ts5 < ts6);

                Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");
            }
        }
    }
```

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

```csharp
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Xunit;
using Platform.Disposables;
using Platform.IO;
using Platform.Ranges;
using Platform.Random;
using Platform.Timestamps;
using Platform.Singletons;
using Platform.Counters;
using Platform.Diagnostics;
using Platform.Data.Constants;
using Platform.Data.Doublets.ResizableDirectMemory;
using Platform.Data.Doublets.Decorators;

namespace Platform.Data.Doublets.Tests
{
    public static class LinksTests
    {
        private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
            Default<LinksCombinedConstants<bool, ulong, int>>.Instance;

        private const long Iterations = 10 * 1024;

        #region Concept

        [Fact]
        public static void MultipleCreateAndDeleteTest()
        {
            //const int N = 21;

            using (var scope = new TempLinksTestScope())
            {
```

```
38              var links = scope.Links;
39
40              for (var N = 0; N < 100; N++)
41              {
42                  var random = new System.Random(N);
43
44                  var created = 0;
45                  var deleted = 0;
46
47                  for (var i = 0; i < N; i++)
48                  {
49                      var linksCount = links.Count();
50
51                      var createPoint = random.NextBoolean();
52
53                      if (linksCount > 2 && createPoint)
54                      {
55                          var linksAddressRange = new Range<ulong>(1, linksCount);
56                          var source = random.NextUInt64(linksAddressRange);
57                          var target = random.NextUInt64(linksAddressRange); //-V3086
58
59                          var resultLink = links.CreateAndUpdate(source, target);
60                          if (resultLink > linksCount)
61                          {
62                              created++;
63                          }
64                      }
65                      else
66                      {
67                          links.Create();
68                          created++;
69                      }
70                  }
71
72                  Assert.True(created == (int)links.Count());
73
74                  for (var i = 0; i < N; i++)
75                  {
76                      var link = (ulong)i + 1;
77                      if (links.Exists(link))
78                      {
79                          links.Delete(link);
80                          deleted++;
81                      }
82                  }
83
84                  Assert.True(links.Count() == 0);
85              }
86          }
87      }
88
89      [Fact]
90      public static void CascadeUpdateTest()
91      {
92          var itself = _constants.Itself;
93
94          using (var scope = new TempLinksTestScope(useLog: true))
95          {
96              var links = scope.Links;
97
98              var l1 = links.Create();
99              var l2 = links.Create();
100
101             l2 = links.Update(l2, l2, l1, l2);
102
103             links.CreateAndUpdate(l2, itself);
104             links.CreateAndUpdate(l2, itself);
105
106             l2 = links.Update(l2, l1);
107
108             links.Delete(l2);
109
110             Global.Trash = links.Count();
111
112             links.Unsync.DisposeIfPossible(); // Close links to access log
113
114             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
    ↪   e.TempTransactionLogFilename);
115         }
116     }
```

```csharp
117
118         [Fact]
119         public static void BasicTransactionLogTest()
120         {
121             using (var scope = new TempLinksTestScope(useLog: true))
122             {
123                 var links = scope.Links;
124                 var l1 = links.Create();
125                 var l2 = links.Create();
126
127                 Global.Trash = links.Update(l2, l2, l1, l2);
128
129                 links.Delete(l1);
130
131                 links.Unsync.DisposeIfPossible(); // Close links to access log
132
133                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
       ↪    e.TempTransactionLogFilename);
134             }
135         }
136
137         [Fact]
138         public static void TransactionAutoRevertedTest()
139         {
140             // Auto Reverted (Because no commit at transaction)
141             using (var scope = new TempLinksTestScope(useLog: true))
142             {
143                 var links = scope.Links;
144                 var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
145                 using (var transaction = transactionsLayer.BeginTransaction())
146                 {
147                     var l1 = links.Create();
148                     var l2 = links.Create();
149
150                     links.Update(l2, l2, l1, l2);
151                 }
152
153                 Assert.Equal(0UL, links.Count());
154
155                 links.Unsync.DisposeIfPossible();
156
157                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
       ↪    cope.TempTransactionLogFilename);
158                 Assert.Single(transitions);
159             }
160         }
161
162         [Fact]
163         public static void TransactionUserCodeErrorNoDataSavedTest()
164         {
165             // User Code Error (Autoreverted), no data saved
166             var itself = _constants.Itself;
167
168             TempLinksTestScope lastScope = null;
169             try
170             {
171                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
       ↪    useLog: true))
172                 {
173                     var links = scope.Links;
174                     var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
       ↪    atorBase<ulong>)links.Unsync).Links;
175                     using (var transaction = transactionsLayer.BeginTransaction())
176                     {
177                         var l1 = links.CreateAndUpdate(itself, itself);
178                         var l2 = links.CreateAndUpdate(itself, itself);
179
180                         l2 = links.Update(l2, l2, l1, l2);
181
182                         links.CreateAndUpdate(l2, itself);
183                         links.CreateAndUpdate(l2, itself);
184
185                         //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
       ↪    tion>(scope.TempTransactionLogFilename);
186
187                         l2 = links.Update(l2, l1);
188
189                         links.Delete(l2);
190
```

```
191                        ExceptionThrower();
192
193                        transaction.Commit();
194                    }
195
196                    Global.Trash = links.Count();
197                }
198            }
199            catch
200            {
201                Assert.False(lastScope == null);
202
203                var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l⌋
    ↪   astScope.TempTransactionLogFilename);
204
205                Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
    ↪   transitions[0].After.IsNull());
206
207                lastScope.DeleteFiles();
208            }
209        }
210
211        [Fact]
212        public static void TransactionUserCodeErrorSomeDataSavedTest()
213        {
214            // User Code Error (Autoreverted), some data saved
215            var itself = _constants.Itself;
216
217            TempLinksTestScope lastScope = null;
218            try
219            {
220                ulong l1;
221                ulong l2;
222
223                using (var scope = new TempLinksTestScope(useLog: true))
224                {
225                    var links = scope.Links;
226                    l1 = links.CreateAndUpdate(itself, itself);
227                    l2 = links.CreateAndUpdate(itself, itself);
228
229                    l2 = links.Update(l2, l2, l1, l2);
230
231                    links.CreateAndUpdate(l2, itself);
232                    links.CreateAndUpdate(l2, itself);
233
234                    links.Unsync.DisposeIfPossible();
235
236                    Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(⌋
    ↪   scope.TempTransactionLogFilename);
237                }
238
239                using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
    ↪   useLog: true))
240                {
241                    var links = scope.Links;
242                    var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
243                    using (var transaction = transactionsLayer.BeginTransaction())
244                    {
245                        l2 = links.Update(l2, l1);
246
247                        links.Delete(l2);
248
249                        ExceptionThrower();
250
251                        transaction.Commit();
252                    }
253
254                    Global.Trash = links.Count();
255                }
256            }
257            catch
258            {
259                Assert.False(lastScope == null);
260
261                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last⌋
    ↪   Scope.TempTransactionLogFilename);
262
263                lastScope.DeleteFiles();
264            }
265        }
```

```csharp
266
267         [Fact]
268         public static void TransactionCommit()
269         {
270             var itself = _constants.Itself;
271
272             var tempDatabaseFilename = Path.GetTempFileName();
273             var tempTransactionLogFilename = Path.GetTempFileName();
274
275             // Commit
276             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                 ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                 ↪  tempTransactionLogFilename))
277             using (var links = new UInt64Links(memoryAdapter))
278             {
279                 using (var transaction = memoryAdapter.BeginTransaction())
280                 {
281                     var l1 = links.CreateAndUpdate(itself, itself);
282                     var l2 = links.CreateAndUpdate(itself, itself);
283
284                     Global.Trash = links.Update(l2, l2, l1, l2);
285
286                     links.Delete(l1);
287
288                     transaction.Commit();
289                 }
290
291                 Global.Trash = links.Count();
292             }
293
294             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                 ↪  sactionLogFilename);
295         }
296
297         [Fact]
298         public static void TransactionDamage()
299         {
300             var itself = _constants.Itself;
301
302             var tempDatabaseFilename = Path.GetTempFileName();
303             var tempTransactionLogFilename = Path.GetTempFileName();
304
305             // Commit
306             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                 ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                 ↪  tempTransactionLogFilename))
307             using (var links = new UInt64Links(memoryAdapter))
308             {
309                 using (var transaction = memoryAdapter.BeginTransaction())
310                 {
311                     var l1 = links.CreateAndUpdate(itself, itself);
312                     var l2 = links.CreateAndUpdate(itself, itself);
313
314                     Global.Trash = links.Update(l2, l2, l1, l2);
315
316                     links.Delete(l1);
317
318                     transaction.Commit();
319                 }
320
321                 Global.Trash = links.Count();
322             }
323
324             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                 ↪  sactionLogFilename);
325
326             // Damage database
327
328             FileHelpers.WriteFirst(tempTransactionLogFilename, new
                 ↪  UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
329
330             // Try load damaged database
331             try
332             {
333                 // TODO: Fix
334                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                     ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                     ↪  tempTransactionLogFilename))
335                 using (var links = new UInt64Links(memoryAdapter))
```

```csharp
                {
                    Global.Trash = links.Count();
                }
            }
            catch (NotSupportedException ex)
            {
                Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
                ↪ yet.");
            }

            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
            ↪ sactionLogFilename);

            File.Delete(tempDatabaseFilename);
            File.Delete(tempTransactionLogFilename);
        }

        [Fact]
        public static void Bug1Test()
        {
            var tempDatabaseFilename = Path.GetTempFileName();
            var tempTransactionLogFilename = Path.GetTempFileName();

            var itself = _constants.Itself;

            // User Code Error (Autoreverted), some data saved
            try
            {
                ulong l1;
                ulong l2;

                using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                ↪ tempTransactionLogFilename))
                using (var links = new UInt64Links(memoryAdapter))
                {
                    l1 = links.CreateAndUpdate(itself, itself);
                    l2 = links.CreateAndUpdate(itself, itself);

                    l2 = links.Update(l2, l2, l1, l2);

                    links.CreateAndUpdate(l2, itself);
                    links.CreateAndUpdate(l2, itself);
                }

                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                ↪ TransactionLogFilename);

                using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                ↪ tempTransactionLogFilename))
                using (var links = new UInt64Links(memoryAdapter))
                {
                    using (var transaction = memoryAdapter.BeginTransaction())
                    {
                        l2 = links.Update(l2, l1);

                        links.Delete(l2);

                        ExceptionThrower();

                        transaction.Commit();
                    }

                    Global.Trash = links.Count();
                }
            }
            catch
            {
                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                ↪ TransactionLogFilename);
            }

            File.Delete(tempDatabaseFilename);
            File.Delete(tempTransactionLogFilename);
        }

        private static void ExceptionThrower()
        {
```

```csharp
                    throw new Exception();
            }

        [Fact]
        public static void PathsTest()
        {
            var source = _constants.SourcePart;
            var target = _constants.TargetPart;

            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var l1 = links.CreatePoint();
                var l2 = links.CreatePoint();

                var r1 = links.GetByKeys(l1, source, target, source);
                var r2 = links.CheckPathExistance(l2, l2, l2, l2);
            }
        }

        [Fact]
        public static void RecursiveStringFormattingTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.

                var a = links.CreatePoint();
                var b = links.CreatePoint();
                var c = links.CreatePoint();

                var ab = links.CreateAndUpdate(a, b);
                var cb = links.CreateAndUpdate(c, b);
                var ac = links.CreateAndUpdate(a, c);

                a = links.Update(a, c, b);
                b = links.Update(b, a, c);
                c = links.Update(c, a, b);

                Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
                Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
                Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));

                Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                    "(5:(4:5 (6:5 4)) 6)");
                Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                    "(6:(5:(4:5 6) 6) 4)");
                Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
                    "(4:(5:4 (6:5 4)) 6)");

                // TODO: Think how to build balanced syntax tree while formatting structure (eg.
                    "(4:(5:4 6) (6:5 4)") instead of "(4:(5:4 (6:5 4)) 6)"

                Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
                    "{{5}{5}{4}{6}}");
                Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                    "{{5}{6}{6}{4}}");
                Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                    "{{4}{5}{4}{6}}");
            }
        }

        private static void DefaultFormatter(StringBuilder sb, ulong link)
        {
            sb.Append(link.ToString());
        }

        #endregion

        #region Performance

        /*
    public static void RunAllPerformanceTests()
    {
        try
        {
            links.TestLinksInSteps();
        }
```

```
479            catch (Exception ex)
480            {
481                ex.WriteToConsole();
482            }
483
484            return;
485
486            try
487            {
488                //ThreadPool.SetMaxThreads(2, 2);
489
490                // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    результат
491                // Также это дополнительно помогает в отладке
492                // Увеличивает вероятность попадания информации в кэши
493                for (var i = 0; i < 10; i++)
494                {
495                    //0 - 10 ГБ
496                    //Каждые 100 МБ срез цифр
497
498                    //links.TestGetSourceFunction();
499                    //links.TestGetSourceFunctionInParallel();
500                    //links.TestGetTargetFunction();
501                    //links.TestGetTargetFunctionInParallel();
502                    links.Create64BillionLinks();
503
504                    links.TestRandomSearchFixed();
505                    //links.Create64BillionLinksInParallel();
506                    links.TestEachFunction();
507                    //links.TestForeach();
508                    //links.TestParallelForeach();
509                }
510
511                links.TestDeletionOfAllLinks();
512
513            }
514            catch (Exception ex)
515            {
516                ex.WriteToConsole();
517            }
518        }*/
519
520         /*
521        public static void TestLinksInSteps()
522        {
523            const long gibibyte = 1024 * 1024 * 1024;
524            const long mebibyte = 1024 * 1024;
525
526            var totalLinksToCreate = gibibyte /
    Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
527            var linksStep = 102 * mebibyte /
    Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
528
529            var creationMeasurements = new List<TimeSpan>();
530            var searchMeasuremets = new List<TimeSpan>();
531            var deletionMeasurements = new List<TimeSpan>();
532
533            GetBaseRandomLoopOverhead(linksStep);
534            GetBaseRandomLoopOverhead(linksStep);
535
536            var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
537
538            ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
539
540            var loops = totalLinksToCreate / linksStep;
541
542            for (int i = 0; i < loops; i++)
543            {
544                creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
545                searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));
546
547                Console.Write("\rC + S {0}/{1}", i + 1, loops);
548            }
549
550            ConsoleHelpers.Debug();
551
552            for (int i = 0; i < loops; i++)
553            {
554                deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
555
```

```csharp
                    Console.Write("\rD {0}/{1}", i + 1, loops);
                }

                ConsoleHelpers.Debug();

                ConsoleHelpers.Debug("C S D");

                for (int i = 0; i < loops; i++)
                {
                    ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
    searchMeasuremets[i], deletionMeasurements[i]);
                }

                ConsoleHelpers.Debug("C S D (no overhead)");

                for (int i = 0; i < loops; i++)
                {
                    ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
    searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
                }

                ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
    links.Total);
            }

        private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
    amountToCreate)
            {
                for (long i = 0; i < amountToCreate; i++)
                    links.Create(0, 0);
            }

            private static TimeSpan GetBaseRandomLoopOverhead(long loops)
            {
                return Measure(() =>
                {
                    ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
                    ulong result = 0;
                    for (long i = 0; i < loops; i++)
                    {
                        var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
                        var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);

                        result += maxValue + source + target;
                    }
                    Global.Trash = result;
                });
            }
             */

        [Fact(Skip = "performance test")]
        public static void GetSourceTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
                    Iterations);

                ulong counter = 0;

                //var firstLink = links.First();
                // Создаём одну связь, из которой будет производить считывание
                var firstLink = links.Create();

                var sw = Stopwatch.StartNew();

                // Тестируем саму функцию
                for (ulong i = 0; i < Iterations; i++)
                {
                    counter += links.GetSource(firstLink);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                // Удаляем связь, из которой производилось считывание
                links.Delete(firstLink);
```

```
                ConsoleHelpers.Debug(
                    "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                    ↪   second), counter result: {3}",
                    Iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

        [Fact(Skip = "performance test")]
        public static void GetSourceInParallel()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
                ↪   parallel.", Iterations);

                long counter = 0;

                //var firstLink = links.First();
                var firstLink = links.Create();

                var sw = Stopwatch.StartNew();

                // Тестируем саму функцию
                Parallel.For(0, Iterations, x =>
                {
                    Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
                    //Interlocked.Increment(ref counter);
                });

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

                ConsoleHelpers.Debug(
                    "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                    ↪   second), counter result: {3}",
                    Iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

        [Fact(Skip = "performance test")]
        public static void TestGetTarget()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
                ↪   Iterations);

                ulong counter = 0;

                //var firstLink = links.First();
                var firstLink = links.Create();

                var sw = Stopwatch.StartNew();

                for (ulong i = 0; i < Iterations; i++)
                {
                    counter += links.GetTarget(firstLink);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

                ConsoleHelpers.Debug(
                    "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                    ↪   second), counter result: {3}",
                    Iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

        [Fact(Skip = "performance test")]
```

```csharp
704         public static void TestGetTargetInParallel()
705         {
706             using (var scope = new TempLinksTestScope())
707             {
708                 var links = scope.Links;
709                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
      ↪      parallel.", Iterations);
710
711                 long counter = 0;
712
713                 //var firstLink = links.First();
714                 var firstLink = links.Create();
715
716                 var sw = Stopwatch.StartNew();
717
718                 Parallel.For(0, Iterations, x =>
719                 {
720                     Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
721                     //Interlocked.Increment(ref counter);
722                 });
723
724                 var elapsedTime = sw.Elapsed;
725
726                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
727
728                 links.Delete(firstLink);
729
730                 ConsoleHelpers.Debug(
731                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
      ↪       second), counter result: {3}",
732                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
733             }
734         }
735
736         // TODO: Заполнить базу данных перед тестом
737         /*
738         [Fact]
739         public void TestRandomSearchFixed()
740         {
741             var tempFilename = Path.GetTempFileName();
742
743             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
      ↪  DefaultLinksSizeStep))
744             {
745                 long iterations = 64 * 1024 * 1024 /
      ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
746
747                 ulong counter = 0;
748                 var maxLink = links.Total;
749
750                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
751
752                 var sw = Stopwatch.StartNew();
753
754                 for (var i = iterations; i > 0; i--)
755                 {
756                     var source =
      ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
757                     var target =
      ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
758
759                     counter += links.Search(source, target);
760                 }
761
762                 var elapsedTime = sw.Elapsed;
763
764                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
765
766                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
      ↪  Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
      ↪  counter);
767             }
768
769             File.Delete(tempFilename);
770         }*/
771
772         [Fact(Skip = "useless: O(0), was dependent on creation tests")]
773         public static void TestRandomSearchAll()
774         {
```

```csharp
                using (var scope = new TempLinksTestScope())
                {
                    var links = scope.Links;
                    ulong counter = 0;

                    var maxLink = links.Count();

                    var iterations = links.Count();

                    ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
                    ↪  links.Count());

                    var sw = Stopwatch.StartNew();

                    for (var i = iterations; i > 0; i--)
                    {
                        var linksAddressRange = new Range<ulong>(_constants.MinPossibleIndex,
                        ↪  maxLink);

                        var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
                        var target = RandomHelpers.Default.NextUInt64(linksAddressRange);

                        counter += links.SearchOrDefault(source, target);
                    }

                    var elapsedTime = sw.Elapsed;

                    var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;

                    ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
                    ↪  Iterations per second), c: {3}",
                        iterations, elapsedTime, (long)iterationsPerSecond, counter);
                }
            }

        [Fact(Skip = "useless: O(0), was dependent on creation tests")]
        public static void TestEach()
        {
                using (var scope = new TempLinksTestScope())
                {
                    var links = scope.Links;

                    var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);

                    ConsoleHelpers.Debug("Testing Each function.");

                    var sw = Stopwatch.StartNew();

                    links.Each(counter.IncrementAndReturnTrue);

                    var elapsedTime = sw.Elapsed;

                    var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;

                    ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
                    ↪  links per second)",
                        counter, elapsedTime, (long)linksPerSecond);
                }
            }

        /*
        [Fact]
        public static void TestForeach()
        {
            var tempFilename = Path.GetTempFileName();

            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪  DefaultLinksSizeStep))
            {
                ulong counter = 0;

                ConsoleHelpers.Debug("Testing foreach through links.");

                var sw = Stopwatch.StartNew();

                //foreach (var link in links)
                //{
                //    counter++;
                //}
```

```
850              var elapsedTime = sw.Elapsed;
851
852              var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
853
854              ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
     ↪  links per second)", counter, elapsedTime, (long)linksPerSecond);
855          }
856
857          File.Delete(tempFilename);
858      }
859      */
860
861      /*
862      [Fact]
863      public static void TestParallelForeach()
864      {
865          var tempFilename = Path.GetTempFileName();
866
867          using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
     ↪  DefaultLinksSizeStep))
868          {
869
870              long counter = 0;
871
872              ConsoleHelpers.Debug("Testing parallel foreach through links.");
873
874              var sw = Stopwatch.StartNew();
875
876              //Parallel.ForEach((IEnumerable<ulong>)links, x =>
877              //{
878              //    Interlocked.Increment(ref counter);
879              //});
880
881              var elapsedTime = sw.Elapsed;
882
883              var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
884
885              ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
     ↪  {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
886          }
887
888          File.Delete(tempFilename);
889      }
890      */
891
892      [Fact(Skip = "performance test")]
893      public static void Create64BillionLinks()
894      {
895          using (var scope = new TempLinksTestScope())
896          {
897              var links = scope.Links;
898              var linksBeforeTest = links.Count();
899
900              long linksToCreate = 64 * 1024 * 1024 /
                 ↪  UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
901
902              ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
903
904              var elapsedTime = Performance.Measure(() =>
905              {
906                  for (long i = 0; i < linksToCreate; i++)
907                  {
908                      links.Create();
909                  }
910              });
911
912              var linksCreated = links.Count() - linksBeforeTest;
913              var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
914
915              ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
916
917              ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                 ↪  linksCreated, elapsedTime,
918                  (long)linksPerSecond);
919          }
920      }
921
922      [Fact(Skip = "performance test")]
923      public static void Create64BillionLinksInParallel()
```

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

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

```
1   using System;
2   using System.Linq;
3   using System.Collections.Generic;
4   using Xunit;
5   using Platform.Interfaces;
6   using Platform.Data.Doublets.Sequences;
7   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
8   using Platform.Data.Doublets.Sequences.Frequencies.Counters;
9   using Platform.Data.Doublets.Sequences.Converters;
10  using Platform.Data.Doublets.PropertyOperators;
11  using Platform.Data.Doublets.Incrementers;
12  using Platform.Data.Doublets.Converters;
13
14  namespace Platform.Data.Doublets.Tests
15  {
16      public static class OptimalVariantSequenceTests
17      {
18          private const string SequenceExample = "зеленела зелёная зелень";
19
20          [Fact]
21          public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
22          {
23              using (var scope = new TempLinksTestScope(useSequences: true))
24              {
25                  var links = scope.Links;
26                  var sequences = scope.Sequences;
27                  var constants = links.Constants;
28
29                  links.UseUnicode();
```

```csharp
30
31                     var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
32
33                     var meaningRoot = links.CreatePoint();
34                     var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
35                     var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
36                     var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
           ↪          constants.Itself);
37
38                     var unaryNumberToAddressConveter = new
           ↪          UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
39                     var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
40                     var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
           ↪          frequencyMarker, unaryOne, unaryNumberIncrementer);
41                     var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
           ↪          frequencyPropertyMarker, frequencyMarker);
42                     var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
           ↪          frequencyPropertyOperator, frequencyIncrementer);
43                     var linkToItsFrequencyNumberConverter = new
           ↪          LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
           ↪          unaryNumberToAddressConveter);
44                     var sequenceToItsLocalElementLevelsConverter = new
           ↪          SequenceToItsLocalElementLevelsConverter<ulong>(links,
           ↪          linkToItsFrequencyNumberConverter);
45                     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
           ↪          sequenceToItsLocalElementLevelsConverter);
46
47                     ExecuteTest(links, sequences, sequence,
           ↪          sequenceToItsLocalElementLevelsConverter, linkFrequencyIncrementer,
           ↪          optimalVariantConverter);
48                 }
49             }
50
51             [Fact]
52             public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
53             {
54                 using (var scope = new TempLinksTestScope(useSequences: true))
55                 {
56                     var links = scope.Links;
57                     var sequences = scope.Sequences;
58
59                     links.UseUnicode();
60
61                     var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
62
63                     var linksToFrequencies = new Dictionary<ulong, ulong>();
64
65                     var totalSequenceSymbolFrequencyCounter = new
           ↪          TotalSequenceSymbolFrequencyCounter<ulong>(links);
66
67                     var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
           ↪          totalSequenceSymbolFrequencyCounter);
68
69                     var linkFrequencyIncrementer = new
           ↪          FrequenciesCacheBasedLinkFrequencyIncrementer<ulong>(linkFrequenciesCache);
70                     var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque⌋
           ↪          ncyNumberConverter<ulong>(linkFrequenciesCache);
71
72                     var sequenceToItsLocalElementLevelsConverter = new
           ↪          SequenceToItsLocalElementLevelsConverter<ulong>(links,
           ↪          linkToItsFrequencyNumberConverter);
73                     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
           ↪          sequenceToItsLocalElementLevelsConverter);
74
75                     ExecuteTest(links, sequences, sequence,
           ↪          sequenceToItsLocalElementLevelsConverter, linkFrequencyIncrementer,
           ↪          optimalVariantConverter);
76                 }
77             }
78
79             private static void ExecuteTest(SynchronizedLinks<ulong> links, Sequences.Sequences
           ↪          sequences, ulong[] sequence, SequenceToItsLocalElementLevelsConverter<ulong>
           ↪          sequenceToItsLocalElementLevelsConverter, IIncrementer<IList<ulong>>
           ↪          linkFrequencyIncrementer, OptimalVariantConverter<ulong> optimalVariantConverter)
80             {
81                 linkFrequencyIncrementer.Increment(sequence);
82
83                 var levels = sequenceToItsLocalElementLevelsConverter.Convert(sequence);
```

```
84
85                    var optimalVariant = optimalVariantConverter.Convert(sequence);
86
87                    var readSequence1 = sequences.ReadSequenceCore(optimalVariant, links.IsPartialPoint);
88
89                    Assert.True(sequence.SequenceEqual(readSequence1));
90                }
91            }
92    }
```

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

```csharp
1    using System;
2    using System.Collections.Generic;
3    using System.Diagnostics;
4    using System.Linq;
5    using Xunit;
6    using Platform.Data.Sequences;
7    using Platform.Data.Doublets.Sequences.Converters;
8
9    namespace Platform.Data.Doublets.Tests
10   {
11       public static class ReadSequenceTests
12       {
13           [Fact]
14           public static void ReadSequenceTest()
15           {
16               const long sequenceLength = 2000;
17
18               using (var scope = new TempLinksTestScope(useSequences: true))
19               {
20                   var links = scope.Links;
21                   var sequences = scope.Sequences;
22
23                   var sequence = new ulong[sequenceLength];
24                   for (var i = 0; i < sequenceLength; i++)
25                   {
26                       sequence[i] = links.Create();
27                   }
28
29                   var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
30
31                   var sw1 = Stopwatch.StartNew();
32                   var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
33
34                   var sw2 = Stopwatch.StartNew();
35                   var readSequence1 = sequences.ReadSequenceCore(balancedVariant,
                     ↪ links.IsPartialPoint); sw2.Stop();
36
37                   var sw3 = Stopwatch.StartNew();
38                   var readSequence2 = new List<ulong>();
39                   SequenceWalker.WalkRight(balancedVariant,
40                                            links.GetSource,
41                                            links.GetTarget,
42                                            links.IsPartialPoint,
43                                            readSequence2.Add);
44                   sw3.Stop();
45
46                   Assert.True(sequence.SequenceEqual(readSequence1));
47
48                   Assert.True(sequence.SequenceEqual(readSequence2));
49
50                   // Assert.True(sw2.Elapsed < sw3.Elapsed);
51
52                   Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
                     ↪ {sw2.Elapsed}");
53
54                   for (var i = 0; i < sequenceLength; i++)
55                   {
56                       links.Delete(sequence[i]);
57                   }
58               }
59           }
60       }
61   }
```

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

```csharp
1    using System.IO;
2    using Xunit;
3    using Platform.Singletons;
4    using Platform.Memory;
```

```csharp
using Platform.Data.Constants;
using Platform.Data.Doublets.ResizableDirectMemory;

namespace Platform.Data.Doublets.Tests
{
    public static class ResizableDirectMemoryLinksTests
    {
        private static readonly LinksCombinedConstants<ulong, ulong, int> _constants =
            Default<LinksCombinedConstants<ulong, ulong, int>>.Instance;

        [Fact]
        public static void BasicFileMappedMemoryTest()
        {
            var tempFilename = Path.GetTempFileName();

            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }

            File.Delete(tempFilename);
        }

        [Fact]
        public static void BasicHeapMemoryTest()
        {
            using (var memory = new
                HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
                UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
        }

        private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Delete(link);
        }

        [Fact]
        public static void NonexistentReferencesHeapMemoryTest()
        {
            using (var memory = new
                HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
                UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestNonexistentReferences();
            }
        }

        private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();

            memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);

            var resultLink = _constants.Null;

            memoryAdapter.Each(foundLink =>
            {
                resultLink = foundLink[_constants.IndexPart];
                return _constants.Break;
            }, _constants.Any, ulong.MaxValue, ulong.MaxValue);

            Assert.True(resultLink == link);

            Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);

            memoryAdapter.Delete(link);
        }
    }
}
```

./Platform.Data.Doublets.Tests/ScopeTests.cs
```csharp
using Xunit;
using Platform.Scopes;
```

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

./Platform.Data.Doublets.Tests/SequencesTests.cs
```
 1   using System;
 2   using System.Collections.Generic;
 3   using System.Diagnostics;
 4   using System.Linq;
 5   using Xunit;
 6   using Platform.Collections;
 7   using Platform.Random;
 8   using Platform.IO;
 9   using Platform.Singletons;
10   using Platform.Data.Constants;
11   using Platform.Data.Doublets.Sequences;
12   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
13   using Platform.Data.Doublets.Sequences.Frequencies.Counters;
14   using Platform.Data.Doublets.Sequences.Converters;
15
16   namespace Platform.Data.Doublets.Tests
17   {
18       public static class SequencesTests
19       {
20           private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
            ↪   Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
21
22           static SequencesTests()
23           {
24               // Trigger static constructor to not mess with perfomance measurements
25               _ = BitString.GetBitMaskFromIndex(1);
26           }
27
28           [Fact]
29           public static void CreateAllVariantsTest()
30           {
31               const long sequenceLength = 8;
32
33               using (var scope = new TempLinksTestScope(useSequences: true))
34               {
35                   var links = scope.Links;
```

```csharp
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var sw1 = Stopwatch.StartNew();
                var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();

                var sw2 = Stopwatch.StartNew();
                var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();

                Assert.True(results1.Count > results2.Length);
                Assert.True(sw1.Elapsed > sw2.Elapsed);

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }

                Assert.True(links.Count() == 0);
            }
        }

        //[Fact]
        //public void CUDTest()
        //{
        //    var tempFilename = Path.GetTempFileName();

        //    const long sequenceLength = 8;

        //    const ulong itself = LinksConstants.Itself;

        //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
        //    DefaultLinksSizeStep))
        //    using (var links = new Links(memoryAdapter))
        //    {
        //        var sequence = new ulong[sequenceLength];
        //        for (var i = 0; i < sequenceLength; i++)
        //            sequence[i] = links.Create(itself, itself);


        //        SequencesOptions o = new SequencesOptions();

        // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
        //        o.

        //        var sequences = new Sequences(links);

        //        var sw1 = Stopwatch.StartNew();
        //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();

        //        var sw2 = Stopwatch.StartNew();
        //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();

        //        Assert.True(results1.Count > results2.Length);
        //        Assert.True(sw1.Elapsed > sw2.Elapsed);

        //        for (var i = 0; i < sequenceLength; i++)
        //            links.Delete(sequence[i]);
        //    }

        //    File.Delete(tempFilename);
        //}

        [Fact]
        public static void AllVariantsSearchTest()
        {
            const long sequenceLength = 8;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
```

```csharp
115              {
116                  sequence[i] = links.Create();
117              }
118
119              var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
120
121              //for (int i = 0; i < createResults.Length; i++)
122              //    sequences.Create(createResults[i]);
123
124              var sw0 = Stopwatch.StartNew();
125              var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
126
127              var sw1 = Stopwatch.StartNew();
128              var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
129
130              var sw2 = Stopwatch.StartNew();
131              var searchResults2 = sequences.Each1(sequence); sw2.Stop();
132
133              var sw3 = Stopwatch.StartNew();
134              var searchResults3 = sequences.Each(sequence); sw3.Stop();
135
136              var intersection0 = createResults.Intersect(searchResults0).ToList();
137              Assert.True(intersection0.Count == searchResults0.Count);
138              Assert.True(intersection0.Count == createResults.Length);
139
140              var intersection1 = createResults.Intersect(searchResults1).ToList();
141              Assert.True(intersection1.Count == searchResults1.Count);
142              Assert.True(intersection1.Count == createResults.Length);
143
144              var intersection2 = createResults.Intersect(searchResults2).ToList();
145              Assert.True(intersection2.Count == searchResults2.Count);
146              Assert.True(intersection2.Count == createResults.Length);
147
148              var intersection3 = createResults.Intersect(searchResults3).ToList();
149              Assert.True(intersection3.Count == searchResults3.Count);
150              Assert.True(intersection3.Count == createResults.Length);
151
152              for (var i = 0; i < sequenceLength; i++)
153              {
154                  links.Delete(sequence[i]);
155              }
156          }
157      }
158
159      [Fact]
160      public static void BalancedVariantSearchTest()
161      {
162          const long sequenceLength = 200;
163
164          using (var scope = new TempLinksTestScope(useSequences: true))
165          {
166              var links = scope.Links;
167              var sequences = scope.Sequences;
168
169              var sequence = new ulong[sequenceLength];
170              for (var i = 0; i < sequenceLength; i++)
171              {
172                  sequence[i] = links.Create();
173              }
174
175              var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
176
177              var sw1 = Stopwatch.StartNew();
178              var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
179
180              var sw2 = Stopwatch.StartNew();
181              var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
182
183              var sw3 = Stopwatch.StartNew();
184              var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
185
186              // На количестве в 200 элементов это будет занимать вечность
187              //var sw4 = Stopwatch.StartNew();
188              //var searchResults4 = sequences.Each(sequence); sw4.Stop();
189
190              Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
191
192              Assert.True(searchResults3.Count == 1 && balancedVariant ==
                  searchResults3.First());
193
```

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

```csharp
                    {
                        var links = scope.Links;
                        var sequences = scope.Sequences;

                        var sequence = new ulong[sequenceLength];
                        for (var i = 0; i < sequenceLength; i++)
                        {
                            sequence[i] = links.Create();
                        }

                        var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

                        var balancedVariant = balancedVariantConverter.Convert(sequence);

                        var partialSequence = new ulong[sequenceLength - 2];

                        Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);

                        var sw1 = Stopwatch.StartNew();
                        var searchResults1 =
                        ↪   sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();

                        var sw2 = Stopwatch.StartNew();
                        var searchResults2 =
                        ↪   sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();

                        Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);

                        Assert.True(searchResults2.Count == 1 && balancedVariant ==
                        ↪   searchResults2.First());

                        for (var i = 0; i < sequenceLength; i++)
                        {
                            links.Delete(sequence[i]);
                        }
                    }
                }

                [Fact(Skip = "Correct implementation is pending")]
                public static void PatternMatchTest()
                {
                    var zeroOrMany = Sequences.Sequences.ZeroOrMany;

                    using (var scope = new TempLinksTestScope(useSequences: true))
                    {
                        var links = scope.Links;
                        var sequences = scope.Sequences;

                        var e1 = links.Create();
                        var e2 = links.Create();

                        var sequence = new[]
                        {
                            e1, e2, e1, e2 // mama / papa
                        };

                        var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

                        var balancedVariant = balancedVariantConverter.Convert(sequence);

                        // 1: [1]
                        // 2: [2]
                        // 3: [1,2]
                        // 4: [1,2,1,2]

                        var doublet = links.GetSource(balancedVariant);

                        var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);

                        Assert.True(matchedSequences1.Count == 0);

                        var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);

                        Assert.True(matchedSequences2.Count == 0);

                        var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);

                        Assert.True(matchedSequences3.Count == 0);

                        var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
```

```
344
345                   Assert.Contains(doublet, matchedSequences4);
346                   Assert.Contains(balancedVariant, matchedSequences4);
347
348                   for (var i = 0; i < sequence.Length; i++)
349                   {
350                       links.Delete(sequence[i]);
351                   }
352               }
353           }
354
355           [Fact]
356           public static void IndexTest()
357           {
358               using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
          ↪    true }, useSequences: true))
359               {
360                   var links = scope.Links;
361                   var sequences = scope.Sequences;
362                   var indexer = sequences.Options.Indexer;
363
364                   var e1 = links.Create();
365                   var e2 = links.Create();
366
367                   var sequence = new[]
368                   {
369                       e1, e2, e1, e2 // mama / papa
370                   };
371
372                   Assert.False(indexer.Index(sequence));
373
374                   Assert.True(indexer.Index(sequence));
375               }
376           }
377
378           /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
          ↪    D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87↲
          ↪    %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
379           private static readonly string _exampleText =
380               @"([english
          ↪    version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
381
382  Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов
  ↪    (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там
  ↪    где есть место для нового начала? Разве пустота это не характеристика пространства?
  ↪    Пространство это то, что можно чем-то наполнить?
383
384  [![чёрное пространство, белое
  ↪    пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
  ↪    ""чёрное пространство, белое пространство"")](https://raw.githubusercontent.com/Konard/Links↲
  ↪    Platform/master/doc/Intro/1.png)
385
386  Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая
  ↪    форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?
387
388  [![чёрное пространство, чёрная
  ↪    точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png
  ↪    ""чёрное пространство, чёрная
  ↪    точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)
389
390  А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
  ↪    так? Инверсия? Отражение? Сумма?
391
392  [![белая точка, чёрная
  ↪    точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png ""белая
  ↪    точка, чёрная
  ↪    точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)
393
394  А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет
  ↪    если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?
  ↪    Гранью? Разделителем? Единицей?
395
396  [![две белые точки, чёрная вертикальная
  ↪    линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png ""две
  ↪    белые точки, чёрная вертикальная
  ↪    линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)
397
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
                        Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
                        ↪   totalCharacters);
                        Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
                        ↪   totalCharacters);

                        Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
                        ↪   totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
                        ↪   totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
                        ↪   totalCharacters}");

                        Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
                        ↪   scope2.Links.Unsync.Count() - initialCount2);
                        Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
                        ↪   scope2.Links.Unsync.Count() - initialCount2);

                        var duplicateProvider1 = new
                        ↪   DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
                        var duplicateProvider2 = new
                        ↪   DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
                        var duplicateProvider3 = new
                        ↪   DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);

                        var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
                        var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
                        var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);

                        var duplicates1 = duplicateCounter1.Count();

                        ConsoleHelpers.Debug("------");

                        var duplicates2 = duplicateCounter2.Count();

                        ConsoleHelpers.Debug("------");

                        var duplicates3 = duplicateCounter3.Count();

                        Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");

                        linkFrequenciesCache1.ValidateFrequencies();
                        linkFrequenciesCache3.ValidateFrequencies();
                    }
            }

        [Fact]
        public static void CompressionStabilityTest()
        {
            // TODO: Fix bug (do a separate test)
            //const ulong minNumbers = 0;
            //const ulong maxNumbers = 1000;

            const ulong minNumbers = 10000;
            const ulong maxNumbers = 12500;

            var strings = new List<string>();

            for (ulong i = minNumbers; i < maxNumbers; i++)
            {
                strings.Add(i.ToString());
            }

            var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
            var totalCharacters = arrays.Select(x => x.Length).Sum();

            using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
            ↪   SequencesOptions<ulong> { UseCompression = true,
            ↪   EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
            using (var scope2 = new TempLinksTestScope(useSequences: true))
            {
                scope1.Links.UseUnicode();
                scope2.Links.UseUnicode();

                //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
                var compressor1 = scope1.Sequences;
                var compressor2 = scope2.Sequences;

                var compressed1 = new ulong[arrays.Length];
                var compressed2 = new ulong[arrays.Length];

                var sw1 = Stopwatch.StartNew();
```

```
var START = 0;
var END = arrays.Length;

// Collisions proved (cannot be solved by max doublet comparison, no stable rule)
// Stability issue starts at 10001 or 11000
//for (int i = START; i < END; i++)
//{
//    var first = compressor1.Compress(arrays[i]);
//    var second = compressor1.Compress(arrays[i]);

//    if (first == second)
//        compressed1[i] = first;
//    else
//    {
//        // TODO: Find a solution for this case
//    }
//}

for (int i = START; i < END; i++)
{
    var first = compressor1.Create(arrays[i]);
    var second = compressor1.Create(arrays[i]);

    if (first == second)
    {
        compressed1[i] = first;
    }
    else
    {
        // TODO: Find a solution for this case
    }
}

var elapsed1 = sw1.Elapsed;

var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);

var sw2 = Stopwatch.StartNew();

for (int i = START; i < END; i++)
{
    var first = balancedVariantConverter.Convert(arrays[i]);
    var second = balancedVariantConverter.Convert(arrays[i]);

    if (first == second)
    {
        compressed2[i] = first;
    }
}

var elapsed2 = sw2.Elapsed;

Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
↪   {elapsed2}");

Assert.True(elapsed1 > elapsed2);

// Checks
for (int i = START; i < END; i++)
{
    var sequence1 = compressed1[i];
    var sequence2 = compressed2[i];

    if (sequence1 != _constants.Null && sequence2 != _constants.Null)
    {
        var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
        ↪   scope1.Links);

        var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
        ↪   scope2.Links);

        //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
        ↪   link.IsPartialPoint());
        //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
        ↪   link.IsPartialPoint());

        //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
        ↪   arrays[i].Length > 3)
```

```
756                    //      Assert.False(structure1 == structure2);

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

```
                var elapsed2 = sw2.Elapsed;

                Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                ↪   {elapsed2}");

                Assert.True(elapsed1 > elapsed2);

                // Checks
                for (int i = START; i < END; i++)
                {
                    var sequence1 = compressed1[i];
                    var sequence2 = compressed2[i];

                    if (sequence1 != _constants.Null && sequence2 != _constants.Null)
                    {
                        var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪   scope1.Links);

                        var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪   scope2.Links);

                        Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
                    }
                }

                Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
                Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);

                Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                ↪   totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                ↪   totalCharacters}");

                // Can be worse than balanced variant
                //Assert.True(scope1.Links.Count() <= scope2.Links.Count());

                //compressor1.ValidateFrequencies();
            }
        }

        [Fact]
        public static void AllTreeBreakDownAtSequencesCreationBugTest()
        {
            // Made out of AllPossibleConnectionsTest test.

            //const long sequenceLength = 5; //100% bug
            const long sequenceLength = 4; //100% bug
            //const long sequenceLength = 3; //100% _no_bug (ok)

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var createResults = sequences.CreateAllVariants2(sequence);

                Global.Trash = createResults;

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }

        [Fact]
        public static void AllPossibleConnectionsTest()
        {
            const long sequenceLength = 5;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;
```

```csharp
                    var sequence = new ulong[sequenceLength];
                    for (var i = 0; i < sequenceLength; i++)
                    {
                        sequence[i] = links.Create();
                    }

                    var createResults = sequences.CreateAllVariants2(sequence);
                    var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());

                    for (var i = 0; i < 1; i++)
                    {
                        var sw1 = Stopwatch.StartNew();
                        var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();

                        var sw2 = Stopwatch.StartNew();
                        var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();

                        var sw3 = Stopwatch.StartNew();
                        var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();

                        var sw4 = Stopwatch.StartNew();
                        var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();

                        Global.Trash = searchResults3;
                        Global.Trash = searchResults4; //-V3008

                        var intersection1 = createResults.Intersect(searchResults1).ToList();
                        Assert.True(intersection1.Count == createResults.Length);

                        var intersection2 = reverseResults.Intersect(searchResults1).ToList();
                        Assert.True(intersection2.Count == reverseResults.Length);

                        var intersection0 = searchResults1.Intersect(searchResults2).ToList();
                        Assert.True(intersection0.Count == searchResults2.Count);

                        var intersection3 = searchResults2.Intersect(searchResults3).ToList();
                        Assert.True(intersection3.Count == searchResults3.Count);

                        var intersection4 = searchResults3.Intersect(searchResults4).ToList();
                        Assert.True(intersection4.Count == searchResults4.Count);
                    }

                    for (var i = 0; i < sequenceLength; i++)
                    {
                        links.Delete(sequence[i]);
                    }
                }
            }

            [Fact(Skip = "Correct implementation is pending")]
            public static void CalculateAllUsagesTest()
            {
                const long sequenceLength = 3;

                using (var scope = new TempLinksTestScope(useSequences: true))
                {
                    var links = scope.Links;
                    var sequences = scope.Sequences;

                    var sequence = new ulong[sequenceLength];
                    for (var i = 0; i < sequenceLength; i++)
                    {
                        sequence[i] = links.Create();
                    }

                    var createResults = sequences.CreateAllVariants2(sequence);

                    //var reverseResults =
                    ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());

                    for (var i = 0; i < 1; i++)
                    {
                        var linksTotalUsages1 = new ulong[links.Count() + 1];

                        sequences.CalculateAllUsages(linksTotalUsages1);

                        var linksTotalUsages2 = new ulong[links.Count() + 1];

                        sequences.CalculateAllUsages2(linksTotalUsages2);
```

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

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

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

Index