LinksPlatform's Platform.Data.Doublets Class Library

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

```csharp
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      public class LinksCascadeUniquenessAndUsagesResolver<TLink> : LinksUniquenessResolver<TLink>
6      {
7          public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLink> links) : base(links) { }
8
9          protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
      ↪   newLinkAddress)
10         {
11             Links.MergeUsages(oldLinkAddress, newLinkAddress);
12             return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
13         }
14     }
15 }
```

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

```csharp
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      /// <remarks>
6      /// <para>Must be used in conjunction with NonNullContentsLinkDeletionResolver.</para>
7      /// <para>Должен использоваться вместе с NonNullContentsLinkDeletionResolver.</para>
8      /// </remarks>
9      public class LinksCascadeUsagesResolver<TLink> : LinksDecoratorBase<TLink>
10     {
11         public LinksCascadeUsagesResolver(ILinks<TLink> links) : base(links) { }
12
13         public override void Delete(TLink linkIndex)
14         {
15             this.DeleteAllUsages(linkIndex);
16             Links.Delete(linkIndex);
17         }
18     }
19 }
```

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

```csharp
1  using System;
2  using System.Collections.Generic;
3  using Platform.Data.Constants;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
10     {
11         public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
12         protected LinksDecoratorBase(ILinks<TLink> links) : base(links) => Constants =
      ↪   links.Constants;
13         public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
14         public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
      ↪   => Links.Each(handler, restrictions);
15         public virtual TLink Create() => Links.Create();
16         public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
17         public virtual void Delete(TLink link) => Links.Delete(link);
18     }
19 }
```

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

```csharp
1  using System;
2  using System.Collections.Generic;
3  using Platform.Disposables;
4  using Platform.Data.Constants;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Decorators
9  {
10     public abstract class LinksDisposableDecoratorBase<TLink> : DisposableBase, ILinks<TLink>
11     {
12         public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
13
14         public ILinks<TLink> Links { get; }
15
```

```
16          protected LinksDisposableDecoratorBase(ILinks<TLink> links)
17          {
18              Links = links;
19              Constants = links.Constants;
20          }
21
22          public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
23
24          public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
            => Links.Each(handler, restrictions);
25
26          public virtual TLink Create() => Links.Create();
27
28          public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
29
30          public virtual void Delete(TLink link) => Links.Delete(link);
31
32          protected override bool AllowMultipleDisposeCalls => true;
33
34          protected override void Dispose(bool manual, bool wasDisposed)
35          {
36              if (!wasDisposed)
37              {
38                  Links.DisposeIfPossible();
39              }
40          }
41      }
42  }
```

./Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs
```
1   using System;
2   using System.Collections.Generic;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Decorators
7   {
8       // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
        be external (hybrid link's raw number).
9       public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
10      {
11          public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
12
13          public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
14          {
15              Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
16              return Links.Each(handler, restrictions);
17          }
18
19          public override TLink Update(IList<TLink> restrictions)
20          {
21              // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
22              Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
23              return Links.Update(restrictions);
24          }
25
26          public override void Delete(TLink link)
27          {
28              Links.EnsureLinkExists(link, nameof(link));
29              Links.Delete(link);
30          }
31      }
32  }
```

./Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs
```
1   using System;
2   using System.Collections.Generic;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Decorators
7   {
8       public class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
9       {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;
11
12          public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
13
14          public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
```

```
15            {
16                var constants = Constants;
17                var itselfConstant = constants.Itself;
18                var indexPartConstant = constants.IndexPart;
19                var sourcePartConstant = constants.SourcePart;
20                var targetPartConstant = constants.TargetPart;
21                var restrictionsCount = restrictions.Count;
22                if (!_equalityComparer.Equals(constants.Any, itselfConstant)
23                 && (((restrictionsCount > indexPartConstant) &&
                 ↪  _equalityComparer.Equals(restrictions[indexPartConstant], itselfConstant))
24                 || ((restrictionsCount > sourcePartConstant) &&
                 ↪  _equalityComparer.Equals(restrictions[sourcePartConstant], itselfConstant))
25                 || ((restrictionsCount > targetPartConstant) &&
                 ↪  _equalityComparer.Equals(restrictions[targetPartConstant], itselfConstant))))
26                {
27                    // Itself constant is not supported for Each method right now, skipping execution
28                    return constants.Continue;
29                }
30                return Links.Each(handler, restrictions);
31            }
32
33            public override TLink Update(IList<TLink> restrictions) =>
             ↪  Links.Update(Links.ResolveConstantAsSelfReference(Constants.Itself, restrictions));
34        }
35    }
```

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

```
1   using System.Collections.Generic;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Decorators
6   {
7       /// <remarks>
8       /// Not practical if newSource and newTarget are too big.
9       /// To be able to use practical version we should allow to create link at any specific
         ↪  location inside ResizableDirectMemoryLinks.
10      /// This in turn will require to implement not a list of empty links, but a list of ranges
         ↪  to store it more efficiently.
11      /// </remarks>
12      public class LinksNonExistentDependenciesCreator<TLink> : LinksDecoratorBase<TLink>
13      {
14          public LinksNonExistentDependenciesCreator(ILinks<TLink> links) : base(links) { }
15
16          public override TLink Update(IList<TLink> restrictions)
17          {
18              var constants = Constants;
19              Links.EnsureCreated(restrictions[constants.SourcePart],
                 ↪  restrictions[constants.TargetPart]);
20              return Links.Update(restrictions);
21          }
22      }
23  }
```

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

```
1   using System.Collections.Generic;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Decorators
6   {
7       public class LinksNullConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
8       {
9           public LinksNullConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
10
11          public override TLink Create()
12          {
13              var link = Links.Create();
14              return Links.Update(link, link, link);
15          }
16
17          public override TLink Update(IList<TLink> restrictions) =>
             ↪  Links.Update(Links.ResolveConstantAsSelfReference(Constants.Null, restrictions));
18      }
19  }
```

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

```csharp
1   using System.Collections.Generic;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Decorators
6   {
7       public class LinksUniquenessResolver<TLink> : LinksDecoratorBase<TLink>
8       {
9           private static readonly EqualityComparer<TLink> _equalityComparer =
                EqualityComparer<TLink>.Default;
10
11          public LinksUniquenessResolver(ILinks<TLink> links) : base(links) { }
12
13          public override TLink Update(IList<TLink> restrictions)
14          {
15              var newLinkAddress = Links.SearchOrDefault(restrictions[Constants.SourcePart],
                    restrictions[Constants.TargetPart]);
16              if (_equalityComparer.Equals(newLinkAddress, default))
17              {
18                  return Links.Update(restrictions);
19              }
20              return ResolveAddressChangeConflict(restrictions[Constants.IndexPart],
                    newLinkAddress);
21          }
22
23          protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
                newLinkAddress)
24          {
25              if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
                    Links.Exists(oldLinkAddress))
26              {
27                  Delete(oldLinkAddress);
28              }
29              return newLinkAddress;
30          }
31      }
32  }
```

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

```csharp
1   using System.Collections.Generic;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Decorators
6   {
7       public class LinksUniquenessValidator<TLink> : LinksDecoratorBase<TLink>
8       {
9           public LinksUniquenessValidator(ILinks<TLink> links) : base(links) { }
10
11          public override TLink Update(IList<TLink> restrictions)
12          {
13              Links.EnsureDoesNotExists(restrictions[Constants.SourcePart],
                    restrictions[Constants.TargetPart]);
14              return Links.Update(restrictions);
15          }
16      }
17  }
```

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

```csharp
1   using System.Collections.Generic;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Decorators
6   {
7       public class LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
8       {
9           public LinksUsagesValidator(ILinks<TLink> links) : base(links) { }
10
11          public override TLink Update(IList<TLink> restrictions)
12          {
13              Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
14              return Links.Update(restrictions);
15          }
16
17          public override void Delete(TLink link)
18          {
19              Links.EnsureNoUsages(link);
```

```
20             Links.Delete(link);
21         }
22     }
23 }
```

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

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      public class NonNullContentsLinkDeletionResolver<TLink> : LinksDecoratorBase<TLink>
6      {
7          public NonNullContentsLinkDeletionResolver(ILinks<TLink> links) : base(links) { }
8
9          public override void Delete(TLink linkIndex)
10         {
11             Links.EnforceResetValues(linkIndex);
12             Links.Delete(linkIndex);
13         }
14     }
15 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using Platform.Collections;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      /// <summary>
10     /// Представляет объект для работы с базой данных (файлом) в формате Links (массива связей).
11     /// </summary>
12     /// <remarks>
13     /// Возможные оптимизации:
14     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
15     ///      + меньше объём БД
16     ///      - меньше производительность
17     ///      - больше ограничение на количество связей в БД)
18     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
19     ///      + меньше объём БД
20     ///      - больше сложность
21     ///
22     /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
23     /// ↪  поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
24     /// ↪  460 752 303 423 488
23     /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
24     /// ↪  (битовыми строками) - вариант матрицы (выстраеваемой лениво).
24     ///
25     /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
26     /// ↪  выбрасываться только при #if DEBUG
26     /// </remarks>
27     public class UInt64Links : LinksDisposableDecoratorBase<ulong>
28     {
29         public UInt64Links(ILinks<ulong> links) : base(links) { }
30
31         public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
32         {
33             this.EnsureLinkIsAnyOrExists(restrictions);
34             return Links.Each(handler, restrictions);
35         }
36
37         public override ulong Create() => Links.CreatePoint();
38
39         public override ulong Update(IList<ulong> restrictions)
40         {
41             var constants = Constants;
42             var nullConstant = constants.Null;
43             if (restrictions.IsNullOrEmpty())
44             {
45                 return nullConstant;
46             }
47             // TODO: Looks like this is a common type of exceptions linked with restrictions
48             // ↪  support
48             if (restrictions.Count != 3)
49             {
50                 throw new NotSupportedException();
51             }
```

```csharp
                var indexPartConstant = constants.IndexPart;
                var updatedLink = restrictions[indexPartConstant];
                this.EnsureLinkExists(updatedLink,
                    $"{nameof(restrictions)}[{nameof(indexPartConstant)}]");
                var sourcePartConstant = constants.SourcePart;
                var newSource = restrictions[sourcePartConstant];
                this.EnsureLinkIsItselfOrExists(newSource,
                    $"{nameof(restrictions)}[{nameof(sourcePartConstant)}]");
                var targetPartConstant = constants.TargetPart;
                var newTarget = restrictions[targetPartConstant];
                this.EnsureLinkIsItselfOrExists(newTarget,
                    $"{nameof(restrictions)}[{nameof(targetPartConstant)}]");
                var existedLink = nullConstant;
                var itselfConstant = constants.Itself;
                if (newSource != itselfConstant && newTarget != itselfConstant)
                {
                    existedLink = this.SearchOrDefault(newSource, newTarget);
                }
                if (existedLink == nullConstant)
                {
                    var before = Links.GetLink(updatedLink);
                    if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
                        newTarget)
                    {
                        Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
                            newSource,
                                                  newTarget == itselfConstant ? updatedLink :
                                                      newTarget);
                    }
                    return updatedLink;
                }
                else
                {
                    return this.MergeAndDelete(updatedLink, existedLink);
                }
            }

            public override void Delete(ulong linkIndex)
            {
                Links.EnsureLinkExists(linkIndex);
                Links.EnforceResetValues(linkIndex);
                this.DeleteAllUsages(linkIndex);
                Links.Delete(linkIndex);
            }
        }
    }
```

./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.Universal;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;
```

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

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

//if (restriction.IsNullOrEmpty()) // Create
//{
//    substitution[Constants.IndexPart] = Memory.AllocateLink();
//    Memory.SetLinkValue(substitution);
//}
//else if (substitution.IsNullOrEmpty()) // Delete
//{
//    Memory.FreeLink(restriction[Constants.IndexPart]);
//}
//else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
//{
//    // No need to collect links to list
//    // Skip == Continue
//    // No need to check substituedHandler
//    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
//  Constants.Break), restriction))
//        return Constants.Break;
//}
//else // Update
//{
//    //List<IList<T>> matchedLinks = null;
//    if (matchedHandler != null)
//    {
//        matchedLinks = new List<IList<T>>();
//        Func<T, bool> handler = link =>
//        {
//            var matchedLink = Memory.GetLinkValue(link);
//            var matchDecision = matchedHandler(matchedLink);
//            if (Equals(matchDecision, Constants.Break))
//                return false;
//            if (!Equals(matchDecision, Constants.Skip))
//                matchedLinks.Add(matchedLink);
//            return true;
//        };
//        if (!Memory.Each(handler, restriction))
//            return Constants.Break;
//    }
//    if (!matchedLinks.IsNullOrEmpty())
//    {
//        var totalMatchedLinks = matchedLinks.Count;
//        for (var i = 0; i < totalMatchedLinks; i++)
//        {
```

```csharp
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();
217                     after[0] = newLink;
218                 }
219                 if (substitution.Count == 1)
220                 {
221                     after = Links.GetLink(substitution[0]);
222                 }
223                 else if (substitution.Count == 3)
224                 {
225                     Links.Update(after);
226                 }
227                 else
228                 {
229                     throw new NotSupportedException();
230                 }
231                 if (matchHandler != null)
232                 {
233                     return substitutionHandler(before, after);
234                 }
235                 return Constants.Continue;
236             }
237             else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
238             {
239                 if (patternOrCondition.Count == 1)
```

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

        /// <remarks>
        /// IList[IList[IList[T]]]
        /// |       |      |    |||
        /// |       |       ------ ||
        /// |       |        link  ||
        /// |       ------------- |
        /// |           change    |
        ///  -------------------
        ///        changes
        /// </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        }
327
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;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets
7  {
8      /// <remarks>
9      /// TODO: Может стоит попробовать ref во всех методах (IRefEqualityComparer)
10     /// 2x faster with comparer
11     /// </remarks>
12     public class DoubletComparer<T> : IEqualityComparer<Doublet<T>>
13     {
14         public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public bool Equals(Doublet<T> x, Doublet<T> y) => x.Equals(y);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public int GetHashCode(Doublet<T> obj) => obj.GetHashCode();
21     }
22 }
```

./Platform.Data.Doublets/Doublet.cs

```
1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets
7  {
8      public struct Doublet<T> : IEquatable<Doublet<T>>
9      {
10         private static readonly EqualityComparer<T> _equalityComparer =
           ↪  EqualityComparer<T>.Default;
11
12         public T Source { get; set; }
13         public T Target { get; set; }
14
15         public Doublet(T source, T target)
16         {
17             Source = source;
18             Target = target;
19         }
20
21         public override string ToString() => $"{Source}->{Target}";
22
23         public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
           ↪  && _equalityComparer.Equals(Target, other.Target);
24
25         public override bool Equals(object obj) => obj is Doublet<T> doublet ?
           ↪  base.Equals(doublet) : false;
26
27         public override int GetHashCode() => (Source, Target).GetHashCode();
28     }
29 }
```

./Platform.Data.Doublets/Hybrid.cs

```
1  using System;
2  using System.Reflection;
3  using Platform.Reflection;
```

```csharp
using Platform.Converters;
using Platform.Exceptions;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        public Hybrid(T value)
        {
            Ensure.Always.IsUnsignedInteger<T>();
            Value = value;
        }

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

        public Hybrid(object value, bool isExternal)
        {
            var signedType = Type<T>.SignedVersion;
            var signedValue = Convert.ChangeType(value, signedType);
            var abs = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Abs").MakeGenericMe
                thod(signedType);
            var negate = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate").MakeGen
                ericMethod(signedType);
            var absoluteValue = abs.Invoke(null, new[] { signedValue });
            var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
                absoluteValue;
            Value = To.UnsignedAs<T>(resultValue);
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

        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);
```

```csharp
            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;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

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

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

        /// <remarks>
```

```csharp
        /// 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];
            //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))
                {
```

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

```csharp
198        /// Возвращает индекс начальной (Source) связи для указанной связи.
199        /// </summary>
200        /// <param name="links">Хранилище связей.</param>
201        /// <param name="link">Связь представленная списком, состоящим из её адреса и
           ↪ содержимого.</param>
202        /// <returns>Индекс начальной связи для указанной связи.</returns>
203        [MethodImpl(MethodImplOptions.AggressiveInlining)]
204        public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
           ↪ link[links.Constants.SourcePart];
205
206        /// <summary>
207        /// Возвращает индекс конечной (Target) связи для указанной связи.
208        /// </summary>
209        /// <param name="links">Хранилище связей.</param>
210        /// <param name="link">Индекс связи.</param>
211        /// <returns>Индекс конечной связи для указанной связи.</returns>
212        [MethodImpl(MethodImplOptions.AggressiveInlining)]
213        public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
           ↪ links.GetLink(link)[links.Constants.TargetPart];
214
215        /// <summary>
216        /// Возвращает индекс конечной (Target) связи для указанной связи.
217        /// </summary>
218        /// <param name="links">Хранилище связей.</param>
219        /// <param name="link">Связь представленная списком, состоящим из её адреса и
           ↪ содержимого.</param>
220        /// <returns>Индекс конечной связи для указанной связи.</returns>
221        [MethodImpl(MethodImplOptions.AggressiveInlining)]
222        public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
           ↪ link[links.Constants.TargetPart];
223
224        /// <summary>
225        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
           ↪ (handler) для каждой подходящей связи.
226        /// </summary>
227        /// <param name="links">Хранилище связей.</param>
228        /// <param name="handler">Обработчик каждой подходящей связи.</param>
229        /// <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>
230        /// <returns>True, в случае если проход по связям не был прерван и False в обратном
           ↪ случае.</returns>
231        [MethodImpl(MethodImplOptions.AggressiveInlining)]
232        public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
           ↪ handler, params TLink[] restrictions)
233            => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
               ↪ links.Constants.Continue);
234
235        /// <summary>
236        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
           ↪ (handler) для каждой подходящей связи.
237        /// </summary>
238        /// <param name="links">Хранилище связей.</param>
239        /// <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>
240        /// <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>
241        /// <param name="handler">Обработчик каждой подходящей связи.</param>
242        /// <returns>True, в случае если проход по связям не был прерван и False в обратном
           ↪ случае.</returns>
243        [MethodImpl(MethodImplOptions.AggressiveInlining)]
244        public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
           ↪ Func<TLink, bool> handler)
245        {
246            var constants = links.Constants;
247            return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
               ↪ constants.Break, constants.Any, source, target);
248        }
249
250        /// <summary>
251        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
           ↪ (handler) для каждой подходящей связи.
252        /// </summary>
253        /// <param name="links">Хранилище связей.</param>
```

```csharp
254        /// <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>
255        /// <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>
256        /// <param name="handler">Обработчик каждой подходящей связи.</param>
257        /// <returns>True, в случае если проход по связям не был прерван и False в обратном
           ↪  случае.</returns>
258        [MethodImpl(MethodImplOptions.AggressiveInlining)]
259        public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
           ↪  Func<IList<TLink>, TLink> handler)
260        {
261            var constants = links.Constants;
262            return links.Each(handler, constants.Any, source, target);
263        }
264
265        [MethodImpl(MethodImplOptions.AggressiveInlining)]
266        public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
           ↪  restrictions)
267        {
268            long arraySize = (Integer<TLink>)links.Count(restrictions);
269            var array = new IList<TLink>[arraySize];
270            if (arraySize > 0)
271            {
272                var filler = new ArrayFiller<IList<TLink>, TLink>(array,
                   ↪  links.Constants.Continue);
273                links.Each(filler.AddAndReturnConstant, restrictions);
274            }
275            return array;
276        }
277
278        [MethodImpl(MethodImplOptions.AggressiveInlining)]
279        public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
           ↪  restrictions)
280        {
281            long arraySize = (Integer<TLink>)links.Count(restrictions);
282            var array = new TLink[arraySize];
283            if (arraySize > 0)
284            {
285                var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
286                links.Each(filler.AddFirstAndReturnConstant, restrictions);
287            }
288            return array;
289        }
290
291        /// <summary>
292        /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
           ↪  в хранилище связей.
293        /// </summary>
294        /// <param name="links">Хранилище связей.</param>
295        /// <param name="source">Начало связи.</param>
296        /// <param name="target">Конец связи.</param>
297        /// <returns>Значение, определяющее существует ли связь.</returns>
298        [MethodImpl(MethodImplOptions.AggressiveInlining)]
299        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;
300
301        #region Ensure
302        // TODO: May be move to EnsureExtensions or make it both there and here
303
304        [MethodImpl(MethodImplOptions.AggressiveInlining)]
305        public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
           ↪  reference, string argumentName)
306        {
307            if (links.IsInnerReference(reference) && !links.Exists(reference))
308            {
309                throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
310            }
311        }
312
313        [MethodImpl(MethodImplOptions.AggressiveInlining)]
314        public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
           ↪  IList<TLink> restrictions, string argumentName)
315        {
316            for (int i = 0; i < restrictions.Count; i++)
```

```csharp
                    {
                        links.EnsureInnerReferenceExists(restrictions[i], argumentName);
                    }
                }

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
                ↪   restrictions)
                {
                    for (int i = 0; i < restrictions.Count; i++)
                    {
                        links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
                    }
                }

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
                ↪   string argumentName)
                {
                    var equalityComparer = EqualityComparer<TLink>.Default;
                    if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
                    {
                        throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
                    }
                }

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
                ↪   link, string argumentName)
                {
                    var equalityComparer = EqualityComparer<TLink>.Default;
                    if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
                    {
                        throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
                    }
                }

                /// <param name="links">Хранилище связей.</param>
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
                ↪   TLink target)
                {
                    if (links.Exists(source, target))
                    {
                        throw new LinkWithSameValueAlreadyExistsException();
                    }
                }

                /// <param name="links">Хранилище связей.</param>
                public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
                {
                    if (links.HasUsages(link))
                    {
                        throw new ArgumentLinkHasDependenciesException<TLink>(link);
                    }
                }

                /// <param name="links">Хранилище связей.</param>
                public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
                ↪   addresses) => links.EnsureCreated(links.Create, addresses);

                /// <param name="links">Хранилище связей.</param>
                public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
                ↪   addresses) => links.EnsureCreated(links.CreatePoint, addresses);

                /// <param name="links">Хранилище связей.</param>
                public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
                ↪   params TLink[] addresses)
                {
                    var constants = links.Constants;
                    var nonExistentAddresses = new HashSet<ulong>(addresses.Where(x =>
                    ↪   !links.Exists(x)).Select(x => (ulong)(Integer<TLink>)x));
                    if (nonExistentAddresses.Count > 0)
                    {
                        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);
```

```csharp
                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 CountUsages<TLink>(this ILinks<TLink> links, TLink link)
        {
            var constants = links.Constants;
            var values = links.GetLink(link);
            ulong usagesAsSource = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
            ↪   link, constants.Any));
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (equalityComparer.Equals(values[constants.SourcePart], link))
            {
                usagesAsSource--;
            }
            ulong usagesAsTarget = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
            ↪   constants.Any, link));
            if (equalityComparer.Equals(values[constants.TargetPart], link))
            {
                usagesAsTarget--;
            }
            return usagesAsSource + usagesAsTarget;
        }

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

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

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

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
{
    var link = links.Create();
    return links.Update(link, link, link);
}

/// <param name="links">Хранилище связей.</param>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
↪   target) => links.Update(links.Create(), source, target);

/// <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="link">Индекс 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 Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
↪   TLink newTarget) => links.Update(new Link<TLink>(link, newSource, newTarget));

/// <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="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>
/// <returns>Индекс обновлённой связи.</returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
{
    if (restrictions.Length == 2)
    {
        return links.MergeAndDelete(restrictions[0], restrictions[1]);
    }
    if (restrictions.Length == 4)
    {
        return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
        ↪   restrictions[2], restrictions[3]);
    }
    else
    {
        return links.Update(restrictions);
    }
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
↪   links, TLink constant, IList<TLink> restrictions)
{
    var equalityComparer = EqualityComparer<TLink>.Default;
    var constants = links.Constants;
    var index = restrictions[constants.IndexPart];
    var source = restrictions[constants.SourcePart];
    var target = restrictions[constants.TargetPart];
    source = equalityComparer.Equals(source, constant) ? index : source;
    target = equalityComparer.Equals(target, constant) ? index : target;
    return new Link<TLink>(index, source, target);
}

/// <summary>
/// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
↪   с указанными Source (началом) и Target (концом).
/// </summary>
/// <param name="links">Хранилище связей.</param>
/// <param name="source">Индекс связи, которая является началом на создаваемой
↪   связи.</param>
/// <param name="target">Индекс связи, которая является концом для создаваемой
↪   связи.</param>
```

```csharp
        /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
        ↪  target)
        {
            var link = links.SearchOrDefault(source, target);
            if (EqualityComparer<TLink>.Default.Equals(link, default))
            {
                link = links.CreateAndUpdate(source, target);
            }
            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]);
            }
        }

        /// <remarks>Before execution of this method ensure that deleted link is detached (all
        ↪  values - source and target are reset to null) or it might enter into infinite
        ↪  recursion.</remarks>
        public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
```

```csharp
590             var anyConstant = links.Constants.Any;
591             var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
592             links.DeleteByQuery(usagesAsSourceQuery);
593             var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
594             links.DeleteByQuery(usagesAsTargetQuery);
595         }
596
597         public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
598         {
599             var count = (Integer<TLink>)links.Count(query);
600             if (count > 0)
601             {
602                 var queryResult = new TLink[count];
603                 var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
                    ↪  links.Constants.Continue);
604                 links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
605                 for (var i = (long)count - 1; i >= 0; i--)
606                 {
607                     links.Delete(queryResult[i]);
608                 }
609             }
610         }
611
612         // TODO: Move to Platform.Data
613         public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
614         {
615             var nullConstant = links.Constants.Null;
616             var equalityComparer = EqualityComparer<TLink>.Default;
617             var link = links.GetLink(linkIndex);
618             for (int i = 1; i < link.Count; i++)
619             {
620                 if (!equalityComparer.Equals(link[i], nullConstant))
621                 {
622                     return false;
623                 }
624             }
625             return true;
626         }
627
628         // TODO: Create a universal version of this method in Platform.Data (with using of for
            ↪  loop)
629         public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
630         {
631             var nullConstant = links.Constants.Null;
632             var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
633             links.Update(updateRequest);
634         }
635
636         // TODO: Create a universal version of this method in Platform.Data (with using of for
            ↪  loop)
637         public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
638         {
639             if (!links.AreValuesReset(linkIndex))
640             {
641                 links.ResetValues(linkIndex);
642             }
643         }
644
645         /// <summary>
646         /// Merging two usages graphs, all children of old link moved to be children of new link
            ↪  or deleted.
647         /// </summary>
648         public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
            ↪  TLink newLinkIndex)
649         {
650             var equalityComparer = EqualityComparer<TLink>.Default;
651             if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
652             {
653                 var constants = links.Constants;
654                 var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
                    ↪  constants.Any);
655                 long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
656                 var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
                    ↪  oldLinkIndex);
657                 long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
658                 var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
                    ↪  usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
659                 if (!isStandalonePoint)
```

```csharp
                    {
                        var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
                        if (totalUsages > 0)
                        {
                            var usages = ArrayPool.Allocate<TLink>(totalUsages);
                            var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
                            ↪  links.Constants.Continue);
                            var i = 0L;
                            if (usagesAsSourceCount > 0)
                            {
                                links.Each(usagesFiller.AddFirstAndReturnConstant,
                                ↪  usagesAsSourceQuery);
                                for (; i < usagesAsSourceCount; i++)
                                {
                                    var usage = usages[i];
                                    if (!equalityComparer.Equals(usage, oldLinkIndex))
                                    {
                                        links.Update(usage, newLinkIndex, links.GetTarget(usage));
                                    }
                                }
                            }
                            if (usagesAsTargetCount > 0)
                            {
                                links.Each(usagesFiller.AddFirstAndReturnConstant,
                                ↪  usagesAsTargetQuery);
                                for (; i < usages.Length; i++)
                                {
                                    var usage = usages[i];
                                    if (!equalityComparer.Equals(usage, oldLinkIndex))
                                    {
                                        links.Update(usage, links.GetSource(usage), newLinkIndex);
                                    }
                                }
                            }
                            ArrayPool.Free(usages);
                        }
                    }
                    return newLinkIndex;
                }

        /// <summary>
        /// Replace one link with another (replaced link is deleted, children are updated or
        ↪  deleted).
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
        ↪  TLink newLinkIndex)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
            {
                links.MergeUsages(oldLinkIndex, newLinkIndex);
                links.Delete(oldLinkIndex);
            }
            return newLinkIndex;
        }
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        private readonly TLink _frequencyMarker;
        private readonly TLink _unaryOne;
        private readonly IIncrementer<TLink> _unaryNumberIncrementer;

        public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
        ↪  IIncrementer<TLink> unaryNumberIncrementer)
```

```
17            : base(links)
18        {
19            _frequencyMarker = frequencyMarker;
20            _unaryOne = unaryOne;
21            _unaryNumberIncrementer = unaryNumberIncrementer;
22        }
23
24        public TLink Increment(TLink frequency)
25        {
26            if (_equalityComparer.Equals(frequency, default))
27            {
28                return Links.GetOrCreate(_unaryOne, _frequencyMarker);
29            }
30            var source = Links.GetSource(frequency);
31            var incrementedSource = _unaryNumberIncrementer.Increment(source);
32            return Links.GetOrCreate(incrementedSource, _frequencyMarker);
33        }
34    }
35 }
```

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

```
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Incrementers
7  {
8      public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
9      {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
              ↪  EqualityComparer<TLink>.Default;
11
12          private readonly TLink _unaryOne;
13
14          public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
              ↪  _unaryOne = unaryOne;
15
16          public TLink Increment(TLink unaryNumber)
17          {
18              if (_equalityComparer.Equals(unaryNumber, _unaryOne))
19              {
20                  return Links.GetOrCreate(_unaryOne, _unaryOne);
21              }
22              var source = Links.GetSource(unaryNumber);
23              var target = Links.GetTarget(unaryNumber);
24              if (_equalityComparer.Equals(source, target))
25              {
26                  return Links.GetOrCreate(unaryNumber, _unaryOne);
27              }
28              else
29              {
30                  return Links.GetOrCreate(source, Increment(target));
31              }
32          }
33      }
34  }
```

./Platform.Data.Doublets/ISynchronizedLinks.cs

```
1  using Platform.Data.Constants;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets
6  {
7      public interface ISynchronizedLinks<TLink> : ISynchronizedLinks<TLink, ILinks<TLink>,
        ↪  LinksCombinedConstants<TLink, TLink, int>>, ILinks<TLink>
8      {
9      }
10 }
```

./Platform.Data.Doublets/Link.cs

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Platform.Exceptions;
5  using Platform.Ranges;
6  using Platform.Singletons;
7  using Platform.Collections.Lists;
```

```csharp
using Platform.Data.Constants;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets
{
    /// <summary>
    /// Структура описывающая уникальную связь.
    /// </summary>
    public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
    {
        public static readonly Link<TLink> Null = new Link<TLink>();

        private static readonly LinksCombinedConstants<bool, TLink, int> _constants =
            Default<LinksCombinedConstants<bool, TLink, int>>.Instance;
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;

        private const int Length = 3;

        public readonly TLink Index;
        public readonly TLink Source;
        public readonly TLink Target;

        public Link(params TLink[] 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 Link(IList<TLink> 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 Link(TLink index, TLink source, TLink target)
        {
            Index = index;
            Source = source;
            Target = target;
        }

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

        public static Link<TLink> Create(TLink source, TLink target) => new Link<TLink>(source,
            target);

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

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

        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();
```

```csharp
        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;
        }

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

        public int IndexOf(TLink item)
        {
            if (_equalityComparer.Equals(Index, item))
            {
                return _constants.IndexPart;
            }
            if (_equalityComparer.Equals(Source, item))
            {
                return _constants.SourcePart;
            }
            if (_equalityComparer.Equals(Target, item))
```

```
151            {
152                return _constants.TargetPart;
153            }
154            return -1;
155        }
156
157        public void Insert(int index, TLink item) => throw new NotSupportedException();
158
159        public void RemoveAt(int index) => throw new NotSupportedException();
160
161        #endregion
162    }
163 }
```

./Platform.Data.Doublets/LinkExtensions.cs

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets
4  {
5      public static class LinkExtensions
6      {
7          public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
         ↪  Point<TLink>.IsFullPoint(link);
8          public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
         ↪  Point<TLink>.IsPartialPoint(link);
9      }
10 }
```

./Platform.Data.Doublets/LinksOperatorBase.cs

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets
4  {
5      public abstract class LinksOperatorBase<TLink>
6      {
7          public ILinks<TLink> Links { get; }
8          protected LinksOperatorBase(ILinks<TLink> links) => Links = links;
9      }
10 }
```

./Platform.Data.Doublets/Numbers/Raw/AddressToRawNumberConverter.cs

```
1  using Platform.Interfaces;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Numbers.Raw
6  {
7      public class AddressToRawNumberConverter<TLink> : IConverter<TLink>
8      {
9          public TLink Convert(TLink source) => new Hybrid<TLink>(source, isExternal: true);
10     }
11 }
```

./Platform.Data.Doublets/Numbers/Raw/RawNumberToAddressConverter.cs

```
1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Numbers.Raw
7  {
8      public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
9      {
10         public TLink Convert(TLink source) => (Integer<TLink>)new
         ↪  Hybrid<TLink>(source).AbsoluteValue;
11     }
12 }
```

./Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs

```
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Reflection;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
```

```
10      public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
        ↪   IConverter<TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪   EqualityComparer<TLink>.Default;
13
14          private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
15
16          public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
            ↪   powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
            ↪   powerOf2ToUnaryNumberConverter;
17
18          public TLink Convert(TLink sourceAddress)
19          {
20              var number = sourceAddress;
21              var nullConstant = Links.Constants.Null;
22              var one = Integer<TLink>.One;
23              var target = nullConstant;
24              for (int i = 0; !_equalityComparer.Equals(number, default) && i <
                ↪   Type<TLink>.BitsLength; i++)
25              {
26                  if (_equalityComparer.Equals(Arithmetic.And(number, one), one))
27                  {
28                      target = _equalityComparer.Equals(target, nullConstant)
29                          ? _powerOf2ToUnaryNumberConverter.Convert(i)
30                          : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
31                  }
32                  number = (Integer<TLink>)((ulong)(Integer<TLink>)number >> 1); // Should be
                    ↪   Bit.ShiftRight(number, 1)
33              }
34              return target;
35          }
36      }
37  }
```

./Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConveter.cs

```
1   using System;
2   using System.Collections.Generic;
3   using Platform.Interfaces;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Numbers.Unary
8   {
9       public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
        ↪   IConverter<Doublet<TLink>, TLink>
10      {
11          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪   EqualityComparer<TLink>.Default;
12
13          private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
14          private readonly IConverter<TLink> _unaryNumberToAddressConverter;
15
16          public LinkToItsFrequencyNumberConveter(
17              ILinks<TLink> links,
18              IPropertyOperator<TLink, TLink> frequencyPropertyOperator,
19              IConverter<TLink> unaryNumberToAddressConverter)
20              : base(links)
21          {
22              _frequencyPropertyOperator = frequencyPropertyOperator;
23              _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
24          }
25
26          public TLink Convert(Doublet<TLink> doublet)
27          {
28              var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
29              if (_equalityComparer.Equals(link, default))
30              {
31                  throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
32              }
33              var frequency = _frequencyPropertyOperator.Get(link);
34              if (_equalityComparer.Equals(frequency, default))
35              {
36                  return default;
37              }
38              var frequencyNumber = Links.GetSource(frequency);
39              return _unaryNumberToAddressConverter.Convert(frequencyNumber);
40          }
41      }
42  }
```

```csharp
1   using System.Collections.Generic;
2   using Platform.Exceptions;
3   using Platform.Interfaces;
4   using Platform.Ranges;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Numbers.Unary
9   {
10      public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<int, TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
13
14          private readonly TLink[] _unaryNumberPowersOf2;
15
16          public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
17          {
18              _unaryNumberPowersOf2 = new TLink[64];
19              _unaryNumberPowersOf2[0] = one;
20          }
21
22          public TLink Convert(int power)
23          {
24              Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
                ↪  - 1), nameof(power));
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  }
```

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Interfaces;
4   using Platform.Numbers;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Numbers.Unary
9   {
10      public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
13
14          private Dictionary<TLink, TLink> _unaryToUInt64;
15          private readonly TLink _unaryOne;
16
17          public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
18              : base(links)
19          {
20              _unaryOne = unaryOne;
21              InitUnaryToUInt64();
22          }
23
24          private void InitUnaryToUInt64()
25          {
26              var one = Integer<TLink>.One;
27              _unaryToUInt64 = new Dictionary<TLink, TLink>
28              {
29                  { _unaryOne, one }
30              };
31              var unary = _unaryOne;
32              var number = one;
33              for (var i = 1; i < 64; i++)
34              {
35                  unary = Links.GetOrCreate(unary, unary);
36                  number = Double(number);
37                  _unaryToUInt64.Add(unary, number);
```

```
38                    }
39                }
40
41            public TLink Convert(TLink unaryNumber)
42            {
43                if (_equalityComparer.Equals(unaryNumber, default))
44                {
45                    return default;
46                }
47                if (_equalityComparer.Equals(unaryNumber, _unaryOne))
48                {
49                    return Integer<TLink>.One;
50                }
51                var source = Links.GetSource(unaryNumber);
52                var target = Links.GetTarget(unaryNumber);
53                if (_equalityComparer.Equals(source, target))
54                {
55                    return _unaryToUInt64[unaryNumber];
56                }
57                else
58                {
59                    var result = _unaryToUInt64[source];
60                    TLink lastValue;
61                    while (!_unaryToUInt64.TryGetValue(target, out lastValue))
62                    {
63                        source = Links.GetSource(target);
64                        result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
65                        target = Links.GetTarget(target);
66                    }
67                    result = Arithmetic<TLink>.Add(result, lastValue);
68                    return result;
69                }
70            }
71
72            [MethodImpl(MethodImplOptions.AggressiveInlining)]
73            private static TLink Double(TLink number) => (Integer<TLink>)((Integer<TLink>)number *
  ↪  2UL);
74        }
75    }
```

./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs

```
1   using System.Collections.Generic;
2   using Platform.Interfaces;
3   using Platform.Reflection;
4   using Platform.Numbers;
5   using System.Runtime.CompilerServices;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Numbers.Unary
10  {
11      public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
  ↪  IConverter<TLink>
12      {
13          private static readonly EqualityComparer<TLink> _equalityComparer =
  ↪  EqualityComparer<TLink>.Default;
14
15          private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
16
17          public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
  ↪  TLink> powerOf2ToUnaryNumberConverter)
18              : base(links)
19          {
20              _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
21              for (int i = 0; i < Type<TLink>.BitsLength; i++)
22              {
23                  _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
24              }
25          }
26
27          public TLink Convert(TLink sourceNumber)
28          {
29              var nullConstant = Links.Constants.Null;
30              var source = sourceNumber;
31              var target = nullConstant;
32              if (!_equalityComparer.Equals(source, nullConstant))
33              {
34                  while (true)
35                  {
36                      if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
```

```
37                          {
38                              SetBit(ref target, powerOf2Index);
39                              break;
40                          }
41                          else
42                          {
43                              powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
44                              SetBit(ref target, powerOf2Index);
45                              source = Links.GetTarget(source);
46                          }
47                      }
48                  }
49                  return target;
50              }
51
52              [MethodImpl(MethodImplOptions.AggressiveInlining)]
53              private static void SetBit(ref TLink target, int powerOf2Index) => target =
            ↪  (Integer<TLink>)((Integer<TLink>)target | 1UL << powerOf2Index); // Should be
            ↪  Math.Or(target, Math.ShiftLeft(One, powerOf2Index))
54          }
55      }
```

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

```
1   using System.Linq;
2   using System.Collections.Generic;
3   using Platform.Interfaces;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.PropertyOperators
8   {
9       public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>,
          ↪  IPropertiesOperator<TLink, TLink, TLink>
10      {
11          private static readonly EqualityComparer<TLink> _equalityComparer =
          ↪  EqualityComparer<TLink>.Default;
12
13          public PropertiesOperator(ILinks<TLink> links) : base(links) { }
14
15          public TLink GetValue(TLink @object, TLink property)
16          {
17              var objectProperty = Links.SearchOrDefault(@object, property);
18              if (_equalityComparer.Equals(objectProperty, default))
19              {
20                  return default;
21              }
22              var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
23              if (valueLink == null)
24              {
25                  return default;
26              }
27              return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
28          }
29
30          public void SetValue(TLink @object, TLink property, TLink value)
31          {
32              var objectProperty = Links.GetOrCreate(@object, property);
33              Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
34              Links.GetOrCreate(objectProperty, value);
35          }
36      }
37  }
```

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

```
1   using System.Collections.Generic;
2   using Platform.Interfaces;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.PropertyOperators
7   {
8       public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IPropertyOperator<TLink,
          ↪  TLink>
9       {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
          ↪  EqualityComparer<TLink>.Default;
11
12          private readonly TLink _propertyMarker;
13          private readonly TLink _propertyValueMarker;
```

```csharp
            public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
            ↪   propertyValueMarker) : base(links)
            {
                _propertyMarker = propertyMarker;
                _propertyValueMarker = propertyValueMarker;
            }

            public TLink Get(TLink link)
            {
                var property = Links.SearchOrDefault(link, _propertyMarker);
                var container = GetContainer(property);
                var value = GetValue(container);
                return value;
            }

            private TLink GetContainer(TLink property)
            {
                var valueContainer = default(TLink);
                if (_equalityComparer.Equals(property, default))
                {
                    return valueContainer;
                }
                var constants = Links.Constants;
                var countinueConstant = constants.Continue;
                var breakConstant = constants.Break;
                var anyConstant = constants.Any;
                var query = new Link<TLink>(anyConstant, property, anyConstant);
                Links.Each(candidate =>
                {
                    var candidateTarget = Links.GetTarget(candidate);
                    var valueTarget = Links.GetTarget(candidateTarget);
                    if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
                    {
                        valueContainer = Links.GetIndex(candidate);
                        return breakConstant;
                    }
                    return countinueConstant;
                }, query);
                return valueContainer;
            }

            private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
            ↪   ? default : Links.GetTarget(container);

            public void Set(TLink link, TLink value)
            {
                var property = Links.GetOrCreate(link, _propertyMarker);
                var container = GetContainer(property);
                if (_equalityComparer.Equals(container, default))
                {
                    Links.GetOrCreate(property, value);
                }
                else
                {
                    Links.Update(container, property, value);
                }
            }
        }
    }
```

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

```csharp
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using Platform.Disposables;
using Platform.Singletons;
using Platform.Collections.Arrays;
using Platform.Numbers;
using Platform.Unsafe;
using Platform.Memory;
using Platform.Data.Exceptions;
using Platform.Data.Constants;
using static Platform.Numbers.Arithmetic;

#pragma warning disable 0649
#pragma warning disable 169
#pragma warning disable 618
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
```

```csharp
19
20  // ReSharper disable StaticMemberInGenericType
21  // ReSharper disable BuiltInTypeReferenceStyle
22  // ReSharper disable MemberCanBePrivate.Local
23  // ReSharper disable UnusedMember.Local
24
25  namespace Platform.Data.Doublets.ResizableDirectMemory
26  {
27      public partial class ResizableDirectMemoryLinks<TLink> : DisposableBase, ILinks<TLink>
28      {
29          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
30          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
31
32          /// <summary>Возвращает размер одной связи в байтах.</summary>
33          public static readonly int LinkSizeInBytes = Structure<Link>.Size;
34
35          public static readonly int LinkHeaderSizeInBytes = Structure<LinksHeader>.Size;
36
37          public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
38
39          private struct Link
40          {
41              public static readonly int SourceOffset = Marshal.OffsetOf(typeof(Link),
                ↪  nameof(Source)).ToInt32();
42              public static readonly int TargetOffset = Marshal.OffsetOf(typeof(Link),
                ↪  nameof(Target)).ToInt32();
43              public static readonly int LeftAsSourceOffset = Marshal.OffsetOf(typeof(Link),
                ↪  nameof(LeftAsSource)).ToInt32();
44              public static readonly int RightAsSourceOffset = Marshal.OffsetOf(typeof(Link),
                ↪  nameof(RightAsSource)).ToInt32();
45              public static readonly int SizeAsSourceOffset = Marshal.OffsetOf(typeof(Link),
                ↪  nameof(SizeAsSource)).ToInt32();
46              public static readonly int LeftAsTargetOffset = Marshal.OffsetOf(typeof(Link),
                ↪  nameof(LeftAsTarget)).ToInt32();
47              public static readonly int RightAsTargetOffset = Marshal.OffsetOf(typeof(Link),
                ↪  nameof(RightAsTarget)).ToInt32();
48              public static readonly int SizeAsTargetOffset = Marshal.OffsetOf(typeof(Link),
                ↪  nameof(SizeAsTarget)).ToInt32();
49
50              public TLink Source;
51              public TLink Target;
52              public TLink LeftAsSource;
53              public TLink RightAsSource;
54              public TLink SizeAsSource;
55              public TLink LeftAsTarget;
56              public TLink RightAsTarget;
57              public TLink SizeAsTarget;
58
59              [MethodImpl(MethodImplOptions.AggressiveInlining)]
60              public static TLink GetSource(IntPtr pointer) => (pointer +
                ↪  SourceOffset).GetValue<TLink>();
61              [MethodImpl(MethodImplOptions.AggressiveInlining)]
62              public static TLink GetTarget(IntPtr pointer) => (pointer +
                ↪  TargetOffset).GetValue<TLink>();
63              [MethodImpl(MethodImplOptions.AggressiveInlining)]
64              public static TLink GetLeftAsSource(IntPtr pointer) => (pointer +
                ↪  LeftAsSourceOffset).GetValue<TLink>();
65              [MethodImpl(MethodImplOptions.AggressiveInlining)]
66              public static TLink GetRightAsSource(IntPtr pointer) => (pointer +
                ↪  RightAsSourceOffset).GetValue<TLink>();
67              [MethodImpl(MethodImplOptions.AggressiveInlining)]
68              public static TLink GetSizeAsSource(IntPtr pointer) => (pointer +
                ↪  SizeAsSourceOffset).GetValue<TLink>();
69              [MethodImpl(MethodImplOptions.AggressiveInlining)]
70              public static TLink GetLeftAsTarget(IntPtr pointer) => (pointer +
                ↪  LeftAsTargetOffset).GetValue<TLink>();
71              [MethodImpl(MethodImplOptions.AggressiveInlining)]
72              public static TLink GetRightAsTarget(IntPtr pointer) => (pointer +
                ↪  RightAsTargetOffset).GetValue<TLink>();
73              [MethodImpl(MethodImplOptions.AggressiveInlining)]
74              public static TLink GetSizeAsTarget(IntPtr pointer) => (pointer +
                ↪  SizeAsTargetOffset).GetValue<TLink>();
75
76              [MethodImpl(MethodImplOptions.AggressiveInlining)]
77              public static void SetSource(IntPtr pointer, TLink value) => (pointer +
                ↪  SourceOffset).SetValue(value);
78              [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

```csharp
133         [MethodImpl(MethodImplOptions.AggressiveInlining)]
134         public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
        ↪   AllocatedLinksOffset).SetValue(value);
135         [MethodImpl(MethodImplOptions.AggressiveInlining)]
136         public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
        ↪   ReservedLinksOffset).SetValue(value);
137         [MethodImpl(MethodImplOptions.AggressiveInlining)]
138         public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
        ↪   FreeLinksOffset).SetValue(value);
139         [MethodImpl(MethodImplOptions.AggressiveInlining)]
140         public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
        ↪   FirstFreeLinkOffset).SetValue(value);
141         [MethodImpl(MethodImplOptions.AggressiveInlining)]
142         public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
        ↪   FirstAsSourceOffset).SetValue(value);
143         [MethodImpl(MethodImplOptions.AggressiveInlining)]
144         public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
        ↪   FirstAsTargetOffset).SetValue(value);
145         [MethodImpl(MethodImplOptions.AggressiveInlining)]
146         public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
        ↪   LastFreeLinkOffset).SetValue(value);
147     }
148
149     private readonly long _memoryReservationStep;
150
151     private readonly IResizableDirectMemory _memory;
152     private IntPtr _header;
153     private IntPtr _links;
154
155     private LinksTargetsTreeMethods _targetsTreeMethods;
156     private LinksSourcesTreeMethods _sourcesTreeMethods;
157
158     // 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
159     private UnusedLinksListMethods _unusedLinksListMethods;
160
161     /// <summary>
162     /// Возвращает общее число связей находящихся в хранилище.
163     /// </summary>
164     private TLink Total => Subtract(LinksHeader.GetAllocatedLinks(_header),
        ↪   LinksHeader.GetFreeLinks(_header));
165
166     public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
167
168     public ResizableDirectMemoryLinks(string address)
169         : this(address, DefaultLinksSizeStep)
170     {
171     }
172
173     /// <summary>
174     /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
        ↪   минимальным шагом расширения базы данных.
175     /// </summary>
176     /// <param name="address">Полный пусть к файлу базы данных.</param>
177     /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
        ↪   байтах.</param>
178     public ResizableDirectMemoryLinks(string address, long memoryReservationStep)
179         : this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
        ↪   memoryReservationStep)
180     {
181     }
182
183     public ResizableDirectMemoryLinks(IResizableDirectMemory memory)
184         : this(memory, DefaultLinksSizeStep)
185     {
186     }
187
188     public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
        ↪   memoryReservationStep)
189     {
190         Constants = Default<LinksCombinedConstants<TLink, TLink, int>>.Instance;
191         _memory = memory;
192         _memoryReservationStep = memoryReservationStep;
193         if (memory.ReservedCapacity < memoryReservationStep)
194         {
195             memory.ReservedCapacity = memoryReservationStep;
196         }
```

```csharp
                SetPointers(_memory);
                // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
                _memory.UsedCapacity = ((long)(Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)
                ↪  * LinkSizeInBytes) + LinkHeaderSizeInBytes;
                // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
                LinksHeader.SetReservedLinks(_header, (Integer<TLink>)((_memory.ReservedCapacity -
                ↪  LinkHeaderSizeInBytes) / LinkSizeInBytes));
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public TLink Count(IList<TLink> 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 (_equalityComparer.Equals(index, Constants.Any))
                    {
                        return Total;
                    }
                    return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
                }
                if (restrictions.Count == 2)
                {
                    var index = restrictions[Constants.IndexPart];
                    var value = restrictions[1];
                    if (_equalityComparer.Equals(index, Constants.Any))
                    {
                        if (_equalityComparer.Equals(value, Constants.Any))
                        {
                            return Total; // Any - как отсутствие ограничения
                        }
                        return Add(_sourcesTreeMethods.CountUsages(value),
                        ↪  _targetsTreeMethods.CountUsages(value));
                    }
                    else
                    {
                        if (!Exists(index))
                        {
                            return Integer<TLink>.Zero;
                        }
                        if (_equalityComparer.Equals(value, Constants.Any))
                        {
                            return Integer<TLink>.One;
                        }
                        var storedLinkValue = GetLinkUnsafe(index);
                        if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
                            _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
                        {
                            return Integer<TLink>.One;
                        }
                        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.CountUsages(target);
                        }
                        else if (_equalityComparer.Equals(target, Constants.Any))
                        {
                            return _sourcesTreeMethods.CountUsages(source);
```

```csharp
                                    }
                                    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)
        {
            if (restrictions.Count == 0)
            {
                for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
                ↪ (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
                ↪ Increment(link))
                {
                    if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
                    ↪ Constants.Break))
                    {
                        return Constants.Break;
                    }
                }

                return Constants.Continue;
            }
            if (restrictions.Count == 1)
            {
                var index = restrictions[Constants.IndexPart];
                if (_equalityComparer.Equals(index, Constants.Any))
                {
                    return Each(handler, ArrayPool<TLink>.Empty);
                }
                if (!Exists(index))
```

```
342                    {
343                        return Constants.Continue;
344                    }
345                    return handler(GetLinkStruct(index));
346                }
347            if (restrictions.Count == 2)
348            {
349                var index = restrictions[Constants.IndexPart];
350                var value = restrictions[1];
351                if (_equalityComparer.Equals(index, Constants.Any))
352                {
353                    if (_equalityComparer.Equals(value, Constants.Any))
354                    {
355                        return Each(handler, ArrayPool<TLink>.Empty);
356                    }
357                    if (_equalityComparer.Equals(Each(handler, new[] { index, value,
      ↪   Constants.Any }), Constants.Break))
358                    {
359                        return Constants.Break;
360                    }
361                    return Each(handler, new[] { index, Constants.Any, value });
362                }
363                else
364                {
365                    if (!Exists(index))
366                    {
367                        return Constants.Continue;
368                    }
369                    if (_equalityComparer.Equals(value, Constants.Any))
370                    {
371                        return handler(GetLinkStruct(index));
372                    }
373                    var storedLinkValue = GetLinkUnsafe(index);
374                    if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
375                        _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
376                    {
377                        return handler(GetLinkStruct(index));
378                    }
379                    return Constants.Continue;
380                }
381            }
382            if (restrictions.Count == 3)
383            {
384                var index = restrictions[Constants.IndexPart];
385                var source = restrictions[Constants.SourcePart];
386                var target = restrictions[Constants.TargetPart];
387                if (_equalityComparer.Equals(index, Constants.Any))
388                {
389                    if (_equalityComparer.Equals(source, Constants.Any) &&
      ↪   _equalityComparer.Equals(target, Constants.Any))
390                    {
391                        return Each(handler, ArrayPool<TLink>.Empty);
392                    }
393                    else if (_equalityComparer.Equals(source, Constants.Any))
394                    {
395                        return _targetsTreeMethods.EachUsage(target, handler);
396                    }
397                    else if (_equalityComparer.Equals(target, Constants.Any))
398                    {
399                        return _sourcesTreeMethods.EachUsage(source, handler);
400                    }
401                    else //if(source != Any && target != Any)
402                    {
403                        var link = _sourcesTreeMethods.Search(source, target);
404                        return _equalityComparer.Equals(link, Constants.Null) ?
      ↪   Constants.Continue : handler(GetLinkStruct(link));
405                    }
406                }
407                else
408                {
409                    if (!Exists(index))
410                    {
411                        return Constants.Continue;
412                    }
413                    if (_equalityComparer.Equals(source, Constants.Any) &&
      ↪   _equalityComparer.Equals(target, Constants.Any))
414                    {
415                        return handler(GetLinkStruct(index));
```

```csharp
                }
                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]);
        Link.SetTarget(link, values[Constants.TargetPart]);
        if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
        {
            _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer(_header),
                linkIndex);
        }
        if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
        {
            _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTargetPointer(_header),
                linkIndex);
        }
        return linkIndex;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public Link<TLink> GetLinkStruct(TLink linkIndex)
    {
        var link = GetLinkUnsafe(linkIndex);
        return new Link<TLink>(linkIndex, Link.GetSource(link), Link.GetTarget(link));
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

```
550                         _unusedLinksListMethods = null;
551                         _targetsTreeMethods = null;
552                         _unusedLinksListMethods = null;
553                     }
554                     else
555                     {
556                         _links = memory.Pointer;
557                         _header = _links;
558                         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
559                         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
560                         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
561                     }
562                 }
563
564                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
565                 private bool Exists(TLink link)
566                     => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
567                     && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
568                     && !IsUnusedLink(link);
569
570                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
571                 private bool IsUnusedLink(TLink link)
572                     => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
573                     || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)),
574                     ↪ Constants.Null)
574                     && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)), Constants.Null));
575
576                 #region DisposableBase
577
578                 protected override bool AllowMultipleDisposeCalls => true;
579
580                 protected override void Dispose(bool manual, bool wasDisposed)
581                 {
582                     if (!wasDisposed)
583                     {
584                         SetPointers(null);
585                         _memory.DisposeIfPossible();
586                     }
587                 }
588
589                 #endregion
590             }
591     }
```

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

```
1   using System;
2   using Platform.Unsafe;
3   using Platform.Collections.Methods.Lists;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.ResizableDirectMemory
8   {
9       partial class ResizableDirectMemoryLinks<TLink>
10      {
11          private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<TLink>
12          {
13              private readonly IntPtr _links;
14              private readonly IntPtr _header;
15
16              public UnusedLinksListMethods(IntPtr links, IntPtr header)
17              {
18                  _links = links;
19                  _header = header;
20              }
21
22              protected override TLink GetFirst() => (_header +
                     ↪ LinksHeader.FirstFreeLinkOffset).GetValue<TLink>();
23
24              protected override TLink GetLast() => (_header +
                     ↪ LinksHeader.LastFreeLinkOffset).GetValue<TLink>();
25
26              protected override TLink GetPrevious(TLink element) =>
                     ↪ (_links.GetElement(LinkSizeInBytes, element) +
                     ↪ Link.SourceOffset).GetValue<TLink>();
27
28              protected override TLink GetNext(TLink element) =>
                     ↪ (_links.GetElement(LinkSizeInBytes, element) +
                     ↪ Link.TargetOffset).GetValue<TLink>();
29
```

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

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

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

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

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

40          protected override void SetSize(TLink size) => (_header +
        ↪   LinksHeader.FreeLinksOffset).SetValue(size);
41      }
42  }
43 }
```

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

```csharp
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;

10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

12 namespace Platform.Data.Doublets.ResizableDirectMemory
13 {
14     partial class ResizableDirectMemoryLinks<TLink>
15     {
16         private abstract class LinksTreeMethodsBase :
            ↪   SizedAndThreadedAVLBalancedTreeMethods<TLink>
17         {
18             private readonly ResizableDirectMemoryLinks<TLink> _memory;
19             private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
20             protected readonly IntPtr Links;
21             protected readonly IntPtr Header;

23             protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
24             {
25                 Links = memory._links;
26                 Header = memory._header;
27                 _memory = memory;
28                 _constants = memory.Constants;
29             }

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

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

37             public TLink this[TLink index]
38             {
39                 get
40                 {
41                     var root = GetTreeRoot();
42                     if (GreaterOrEqualThan(index, GetSize(root)))
43                     {
44                         return GetZero();
45                     }
46                     while (!EqualToZero(root))
47                     {
48                         var left = GetLeftOrDefault(root);
49                         var leftSize = GetSizeOrZero(left);
50                         if (LessThan(index, leftSize))
51                         {
52                             root = left;
53                             continue;
54                         }
55                         if (IsEquals(index, leftSize))
56                         {
```

```
57                        return root;
58                    }
59                    root = GetRightOrDefault(root);
60                    index = Subtract(index, Increment(leftSize));
61                }
62                return GetZero(); // TODO: Impossible situation exception (only if tree
   ↳    structure broken)
63            }
64        }
65
66        // TODO: Return indices range instead of references count
67        public TLink CountUsages(TLink link)
68        {
69            var root = GetTreeRoot();
70            var total = GetSize(root);
71            var totalRightIgnore = GetZero();
72            while (!EqualToZero(root))
73            {
74                var @base = GetBasePartValue(root);
75                if (LessOrEqualThan(@base, link))
76                {
77                    root = GetRightOrDefault(root);
78                }
79                else
80                {
81                    totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
82                    root = GetLeftOrDefault(root);
83                }
84            }
85            root = GetTreeRoot();
86            var totalLeftIgnore = GetZero();
87            while (!EqualToZero(root))
88            {
89                var @base = GetBasePartValue(root);
90                if (GreaterOrEqualThan(@base, link))
91                {
92                    root = GetLeftOrDefault(root);
93                }
94                else
95                {
96                    totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
97
98                    root = GetRightOrDefault(root);
99                }
100           }
101           return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
102       }
103
104       public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
105       {
106           var root = GetTreeRoot();
107           if (EqualToZero(root))
108           {
109               return _constants.Continue;
110           }
111           TLink first = GetZero(), current = root;
112           while (!EqualToZero(current))
113           {
114               var @base = GetBasePartValue(current);
115               if (GreaterOrEqualThan(@base, link))
116               {
117                   if (IsEquals(@base, link))
118                   {
119                       first = current;
120                   }
121                   current = GetLeftOrDefault(current);
122               }
123               else
124               {
125                   current = GetRightOrDefault(current);
126               }
127           }
128           if (!EqualToZero(first))
129           {
130               current = first;
131               while (true)
132               {
133                   if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
```

```
134                                {
135                                    return _constants.Break;
136                                }
137                                current = GetNext(current);
138                                if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
139                                {
140                                    break;
141                                }
142                            }
143                        }
144                        return _constants.Continue;
145                    }
146
147            protected override void PrintNodeValue(TLink node, StringBuilder sb)
148            {
149                sb.Append(' ');
150                sb.Append((Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.SourceOffset).GetValue<TLink>());
151                sb.Append('-');
152                sb.Append('>');
153                sb.Append((Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.TargetOffset).GetValue<TLink>());
154            }
155        }
156
157        private class LinksSourcesTreeMethods : LinksTreeMethodsBase
158        {
159            public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
160                : base(memory)
161            {
162            }
163
164            protected override IntPtr GetLeftPointer(TLink node) =>
                ↪  Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset;
165
166            protected override IntPtr GetRightPointer(TLink node) =>
                ↪  Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset;
167
168            protected override TLink GetLeftValue(TLink node) =>
                ↪  (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.LeftAsSourceOffset).GetValue<TLink>();
169
170            protected override TLink GetRightValue(TLink node) =>
                ↪  (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.RightAsSourceOffset).GetValue<TLink>();
171
172            protected override TLink GetSize(TLink node)
173            {
174                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
175                return Bit.PartialRead(previousValue, 5, -5);
176            }
177
178            protected override void SetLeft(TLink node, TLink left) =>
                ↪  (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.LeftAsSourceOffset).SetValue(left);
179
180            protected override void SetRight(TLink node, TLink right) =>
                ↪  (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.RightAsSourceOffset).SetValue(right);
181
182            protected override void SetSize(TLink node, TLink size)
183            {
184                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
185                (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
                    ↪  -5));
186            }
187
188            protected override bool GetLeftIsChild(TLink node)
189            {
190                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
191                return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
192            }
193
194            protected override void SetLeftIsChild(TLink node, bool value)
```

```csharp
                    {
                        var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                        ↪   Link.SizeAsSourceOffset).GetValue<TLink>();
                        var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
                        ↪   1);
                        (Links.GetElement(LinkSizeInBytes, node) +
                        ↪   Link.SizeAsSourceOffset).SetValue(modified);
                    }

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

                    protected override void SetRightIsChild(TLink node, bool value)
                    {
                        var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                        ↪   Link.SizeAsSourceOffset).GetValue<TLink>();
                        var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
                        ↪   1);
                        (Links.GetElement(LinkSizeInBytes, node) +
                        ↪   Link.SizeAsSourceOffset).SetValue(modified);
                    }

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

                    protected override void SetBalance(TLink node, sbyte value)
                    {
                        var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                        ↪   Link.SizeAsSourceOffset).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.SizeAsSourceOffset).SetValue(modified);
                    }

                    protected override bool FirstIsToTheLeftOfSecond(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 LessThan(firstSource, secondSource) ||
                                (IsEquals(firstSource, secondSource) &&
                                    LessThan((Links.GetElement(LinkSizeInBytes, first) +
                        ↪   Link.TargetOffset).GetValue<TLink>(),
                        ↪   (Links.GetElement(LinkSizeInBytes, second) +
                        ↪   Link.TargetOffset).GetValue<TLink>()));
                    }

                    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>()));
                    }
```

```csharp
        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)
        {
        }

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

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

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

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

        protected override TLink GetSize(TLink node)
        {
```

```csharp
            var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪   Link.SizeAsTargetOffset).GetValue<TLink>();
            return Bit.PartialRead(previousValue, 5, -5);
        }

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

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

        protected override void SetSize(TLink node, TLink size)
        {
            var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪   Link.SizeAsTargetOffset).GetValue<TLink>();
            (Links.GetElement(LinkSizeInBytes, node) +
↪   Link.SizeAsTargetOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
↪   -5));
        }

        protected override bool GetLeftIsChild(TLink node)
        {
            var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪   Link.SizeAsTargetOffset).GetValue<TLink>();
            return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
        }

        protected override void SetLeftIsChild(TLink node, bool value)
        {
            var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪   Link.SizeAsTargetOffset).GetValue<TLink>();
            var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
↪   1);
            (Links.GetElement(LinkSizeInBytes, node) +
↪   Link.SizeAsTargetOffset).SetValue(modified);
        }

        protected override bool GetRightIsChild(TLink node)
        {
            var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪   Link.SizeAsTargetOffset).GetValue<TLink>();
            return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
        }

        protected override void SetRightIsChild(TLink node, bool value)
        {
            var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
↪   Link.SizeAsTargetOffset).GetValue<TLink>();
            var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
↪   1);
            (Links.GetElement(LinkSizeInBytes, node) +
↪   Link.SizeAsTargetOffset).SetValue(modified);
        }

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

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

```csharp
                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;

#pragma warning disable 0649
#pragma warning disable 169
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

// ReSharper disable BuiltInTypeReferenceStyle

//#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION

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>
        /// 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;
```

```csharp
            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
        //   байтах.</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);
            // 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;
```

```csharp
                    }
                    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.CountUsages(value)
                                    + _targetsTreeMethods.CountUsages(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.CountUsages(target);
                            }
                            else if (target == Constants.Any)
                            {
                                return _sourcesTreeMethods.CountUsages(source);
                            }
                            else //if(source != Any && target != Any)
                            {
                                // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
                                var link = _sourcesTreeMethods.Search(source, target);
                                return link == Constants.Null ? 0UL : 1UL;
                            }
                        }
                        else
                        {
                            if (!Exists(index))
                            {
                                return 0;
                            }
                            if (source == Constants.Any && target == Constants.Any)
                            {
                                return 1;
                            }
```

```
190                      var storedLinkValue = GetLinkUnsafe(index);
191                      if (source != Constants.Any && target != Constants.Any)
192                      {
193                          if (storedLinkValue->Source == source &&
194                              storedLinkValue->Target == target)
195                          {
196                              return 1;
197                          }
198                          return 0;
199                      }
200                      var value = default(id);
201                      if (source == Constants.Any)
202                      {
203                          value = target;
204                      }
205                      if (target == Constants.Any)
206                      {
207                          value = source;
208                      }
209                      if (storedLinkValue->Source == value ||
210                          storedLinkValue->Target == value)
211                      {
212                          return 1;
213                      }
214                      return 0;
215                  }
216              }
217              throw new NotSupportedException("Другие размеры и способы ограничений не
                 ↪ поддерживаются.");
218          }
219
220          [MethodImpl(MethodImplOptions.AggressiveInlining)]
221          public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
222          {
223              if (restrictions.Count == 0)
224              {
225                  for (id link = 1; link <= _header->AllocatedLinks; link++)
226                  {
227                      if (Exists(link))
228                      {
229                          if (handler(GetLinkStruct(link)) == Constants.Break)
230                          {
231                              return Constants.Break;
232                          }
233                      }
234                  }
235                  return Constants.Continue;
236              }
237              if (restrictions.Count == 1)
238              {
239                  var index = restrictions[Constants.IndexPart];
240                  if (index == Constants.Any)
241                  {
242                      return Each(handler, ArrayPool<ulong>.Empty);
243                  }
244                  if (!Exists(index))
245                  {
246                      return Constants.Continue;
247                  }
248                  return handler(GetLinkStruct(index));
249              }
250              if (restrictions.Count == 2)
251              {
252                  var index = restrictions[Constants.IndexPart];
253                  var value = restrictions[1];
254                  if (index == Constants.Any)
255                  {
256                      if (value == Constants.Any)
257                      {
258                          return Each(handler, ArrayPool<ulong>.Empty);
259                      }
260                      if (Each(handler, new[] { index, value, Constants.Any }) == Constants.Break)
261                      {
262                          return Constants.Break;
263                      }
264                      return Each(handler, new[] { index, Constants.Any, value });
265                  }
266                  else
```

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

```csharp
                        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];

        /// <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
            {
```

```csharp
                    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);
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool Exists(id link) => link >= Constants.MinPossibleIndex && link <=
             _header->AllocatedLinks && !IsUnusedLink(link);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
                                        || (_links[link].SizeAsSource == Constants.Null &&
                                         _links[link].Source != Constants.Null);

            #region Disposable

            protected override bool AllowMultipleDisposeCalls => true;
```

```
493
494        protected override void Dispose(bool manual, bool wasDisposed)
495        {
496            if (!wasDisposed)
497            {
498                SetPointers(null);
499                _memory.DisposeIfPossible();
500            }
501        }
502
503        #endregion
504    }
505 }
```

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

```
1  using Platform.Collections.Methods.Lists;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.ResizableDirectMemory
6  {
7      unsafe partial class UInt64ResizableDirectMemoryLinks
8      {
9          private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<ulong>
10         {
11             private readonly Link* _links;
12             private readonly LinksHeader* _header;
13
14             public UnusedLinksListMethods(Link* links, LinksHeader* header)
15             {
16                 _links = links;
17                 _header = header;
18             }
19
20             protected override ulong GetFirst() => _header->FirstFreeLink;
21
22             protected override ulong GetLast() => _header->LastFreeLink;
23
24             protected override ulong GetPrevious(ulong element) => _links[element].Source;
25
26             protected override ulong GetNext(ulong element) => _links[element].Target;
27
28             protected override ulong GetSize() => _header->FreeLinks;
29
30             protected override void SetFirst(ulong element) => _header->FirstFreeLink = element;
31
32             protected override void SetLast(ulong element) => _header->LastFreeLink = element;
33
34             protected override void SetPrevious(ulong element, ulong previous) =>
                  _links[element].Source = previous;
35
36             protected override void SetNext(ulong element, ulong next) => _links[element].Target
                  = next;
37
38             protected override void SetSize(ulong size) => _header->FreeLinks = size;
39         }
40     }
41 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using System.Text;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Data.Constants;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory
11 {
12     unsafe partial class UInt64ResizableDirectMemoryLinks
13     {
14         private abstract class LinksTreeMethodsBase :
                SizedAndThreadedAVLBalancedTreeMethods<ulong>
15         {
16             private readonly UInt64ResizableDirectMemoryLinks _memory;
17             private readonly LinksCombinedConstants<ulong, ulong, int> _constants;
18             protected readonly Link* Links;
19             protected readonly LinksHeader* Header;
20
```

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

```csharp
                    }

        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;

        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;
        }
```

```csharp
        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);

        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>
```

```csharp
        /// 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)]
        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;
```

```csharp
            [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;

            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;
            }
```

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

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

387             protected override void SetSize(ulong node, ulong size)
388             {
389                 var previousValue = Links[node].SizeAsTarget;
390                 //var modified = Math.PartialWrite(previousValue, size, 5, -5);
391                 var modified = (previousValue & 31) | ((size & 134217727) << 5);
392                 Links[node].SizeAsTarget = modified;
393             }

395             protected override bool GetLeftIsChild(ulong node)
396             {
397                 var previousValue = Links[node].SizeAsTarget;
398                 //return (Integer)Math.PartialRead(previousValue, 4, 1);
399                 return (previousValue & 16) >> 4 == 1UL;
400                 // TODO: Check if this is possible to use
401                 //var nodeSize = GetSize(node);
402                 //var left = GetLeftValue(node);
403                 //var leftSize = GetSizeOrZero(left);
404                 //return leftSize > 0 && nodeSize > leftSize;
405             }

407             protected override void SetLeftIsChild(ulong node, bool value)
408             {
409                 var previousValue = Links[node].SizeAsTarget;
410                 //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
411                 var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
412                 Links[node].SizeAsTarget = modified;
413             }

415             protected override bool GetRightIsChild(ulong node)
416             {
417                 var previousValue = Links[node].SizeAsTarget;
418                 //return (Integer)Math.PartialRead(previousValue, 3, 1);
419                 return (previousValue & 8) >> 3 == 1UL;
420                 // TODO: Check if this is possible to use
421                 //var nodeSize = GetSize(node);
422                 //var right = GetRightValue(node);
423                 //var rightSize = GetSizeOrZero(right);
424                 //return rightSize > 0 && nodeSize > rightSize;
425             }

427             protected override void SetRightIsChild(ulong node, bool value)
428             {
429                 var previousValue = Links[node].SizeAsTarget;
430                 //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
431                 var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
432                 Links[node].SizeAsTarget = modified;
433             }

435             protected override sbyte GetBalance(ulong node)
436             {
437                 var previousValue = Links[node].SizeAsTarget;
438                 //var value = Math.PartialRead(previousValue, 0, 3);
439                 var value = previousValue & 7;
440                 var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
        ↪   124 : value & 3);
441                 return unpackedValue;
442             }

444             protected override void SetBalance(ulong node, sbyte value)
445             {
446                 var previousValue = Links[node].SizeAsTarget;
447                 var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
448                 //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
449                 var modified = (previousValue & 4294967288) | (packagedValue & 7);
450                 Links[node].SizeAsTarget = modified;
451             }

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

```
457          protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
458              => Links[first].Target > Links[second].Target ||
459                  (Links[first].Target == Links[second].Target && Links[first].Source >
                     ↪  Links[second].Source);
460
461          protected override ulong GetTreeRoot() => Header->FirstAsTarget;
462
463          protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
464
465          [MethodImpl(MethodImplOptions.AggressiveInlining)]
466          protected override void ClearNode(ulong node)
467          {
468              Links[node].LeftAsTarget = 0UL;
469              Links[node].RightAsTarget = 0UL;
470              Links[node].SizeAsTarget = 0UL;
471          }
472      }
473    }
474  }
```

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

```
1   using System.Collections.Generic;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Sequences.Converters
6   {
7       public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
8       {
9           public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
10
11          public override TLink Convert(IList<TLink> sequence)
12          {
13              var length = sequence.Count;
14              if (length < 1)
15              {
16                  return default;
17              }
18              if (length == 1)
19              {
20                  return sequence[0];
21              }
22              // Make copy of next layer
23              if (length > 2)
24              {
25                  // TODO: Try to use stackalloc (which at the moment is not working with
                     ↪  generics) but will be possible with Sigil
26                  var halvedSequence = new TLink[(length / 2) + (length % 2)];
27                  HalveSequence(halvedSequence, sequence, length);
28                  sequence = halvedSequence;
29                  length = halvedSequence.Length;
30              }
31              // Keep creating layer after layer
32              while (length > 2)
33              {
34                  HalveSequence(sequence, sequence, length);
35                  length = (length / 2) + (length % 2);
36              }
37              return Links.GetOrCreate(sequence[0], sequence[1]);
38          }
39
40          private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
41          {
42              var loopedLength = length - (length % 2);
43              for (var i = 0; i < loopedLength; i += 2)
44              {
45                  destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
46              }
47              if (length > loopedLength)
48              {
49                  destination[length / 2] = source[length - 1];
50              }
51          }
52      }
53  }
```

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

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

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

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

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

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

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

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

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

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

```csharp
72
73          /// <remarks>
74          /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
75          /// Faster version (doublets' frequencies dictionary is not recreated).
76          /// </remarks>
77          private IList<TLink> Compress(IList<TLink> sequence)
78          {
79              if (sequence.IsNullOrEmpty())
80              {
81                  return null;
82              }
83              if (sequence.Count == 1)
84              {
85                  return sequence;
86              }
87              if (sequence.Count == 2)
88              {
89                  return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
90              }
91              // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
92              var copy = new HalfDoublet[sequence.Count];
93              Doublet<TLink> doublet = default;
94              for (var i = 1; i < sequence.Count; i++)
95              {
96                  doublet.Source = sequence[i - 1];
97                  doublet.Target = sequence[i];
98                  LinkFrequency<TLink> data;
99                  if (_doInitialFrequenciesIncrement)
100                 {
101                     data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
102                 }
103                 else
104                 {
105                     data = _doubletFrequenciesCache.GetFrequency(ref doublet);
106                     if (data == null)
107                     {
108                         throw new NotSupportedException("If you ask not to increment
                    ↪  frequencies, it is expected that all frequencies for the sequence
                    ↪  are prepared.");
109                     }
110                 }
111                 copy[i - 1].Element = sequence[i - 1];
112                 copy[i - 1].DoubletData = data;
113                 UpdateMaxDoublet(ref doublet, data);
114             }
115             copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
116             copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
117             if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
118             {
119                 var newLength = ReplaceDoublets(copy);
120                 sequence = new TLink[newLength];
121                 for (int i = 0; i < newLength; i++)
122                 {
123                     sequence[i] = copy[i].Element;
124                 }
125             }
126             return sequence;
127         }
128
129         /// <remarks>
130         /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
131         /// </remarks>
132         private int ReplaceDoublets(HalfDoublet[] copy)
133         {
134             var oldLength = copy.Length;
135             var newLength = copy.Length;
136             while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
137             {
138                 var maxDoubletSource = _maxDoublet.Source;
139                 var maxDoubletTarget = _maxDoublet.Target;
140                 if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
141                 {
142                     _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
143                 }
144                 var maxDoubletReplacementLink = _maxDoubletData.Link;
145                 oldLength--;
146                 var oldLengthMinusTwo = oldLength - 1;
147                 // Substitute all usages
148                 int w = 0, r = 0; // (r == read, w == write)
```

```
149                    for (; r < oldLength; r++)
150                    {
151                        if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                               ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
152                        {
153                            if (r > 0)
154                            {
155                                var previous = copy[w - 1].Element;
156                                copy[w - 1].DoubletData.DecrementFrequency();
157                                copy[w - 1].DoubletData =
                                   ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
                                   ↪ maxDoubletReplacementLink);
158                            }
159                            if (r < oldLengthMinusTwo)
160                            {
161                                var next = copy[r + 2].Element;
162                                copy[r + 1].DoubletData.DecrementFrequency();
163                                copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma↲
                                   ↪ xDoubletReplacementLink,
                                   ↪ next);
164                            }
165                            copy[w++].Element = maxDoubletReplacementLink;
166                            r++;
167                            newLength--;
168                        }
169                        else
170                        {
171                            copy[w++] = copy[r];
172                        }
173                    }
174                    if (w < newLength)
175                    {
176                        copy[w] = copy[r];
177                    }
178                    oldLength = newLength;
179                    ResetMaxDoublet();
180                    UpdateMaxDoublet(copy, newLength);
181                }
182                return newLength;
183            }
184
185            [MethodImpl(MethodImplOptions.AggressiveInlining)]
186            private void ResetMaxDoublet()
187            {
188                _maxDoublet = new Doublet<TLink>();
189                _maxDoubletData = new LinkFrequency<TLink>();
190            }
191
192            [MethodImpl(MethodImplOptions.AggressiveInlining)]
193            private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
194            {
195                Doublet<TLink> doublet = default;
196                for (var i = 1; i < length; i++)
197                {
198                    doublet.Source = copy[i - 1].Element;
199                    doublet.Target = copy[i].Element;
200                    UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
201                }
202            }
203
204            [MethodImpl(MethodImplOptions.AggressiveInlining)]
205            private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
206            {
207                var frequency = data.Frequency;
208                var maxFrequency = _maxDoubletData.Frequency;
209                //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
                   ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
                   ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
                   ↪ _maxDoublet.Target)))
210                if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
211                    (_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) */
212                {
213                    _maxDoublet = doublet;
```

```
214            _maxDoubletData = data;
215        }
216    }
217 }
218 }
```

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

```csharp
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Converters
7  {
8      public abstract class LinksListToSequenceConverterBase<TLink> : IConverter<IList<TLink>,
         ↪  TLink>
9      {
10         protected readonly ILinks<TLink> Links;
11         public LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
12         public abstract TLink Convert(IList<TLink> source);
13     }
14 }
```

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

```csharp
1  using System.Collections.Generic;
2  using System.Linq;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.Converters
8  {
9      public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
12         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13
14         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
15
16         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
            ↪  sequenceToItsLocalElementLevelsConverter) : base(links)
17             => _sequenceToItsLocalElementLevelsConverter =
                ↪  sequenceToItsLocalElementLevelsConverter;
18
19         public override TLink Convert(IList<TLink> sequence)
20         {
21             var length = sequence.Count;
22             if (length == 1)
23             {
24                 return sequence[0];
25             }
26             var links = Links;
27             if (length == 2)
28             {
29                 return links.GetOrCreate(sequence[0], sequence[1]);
30             }
31             sequence = sequence.ToArray();
32             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
33             while (length > 2)
34             {
35                 var levelRepeat = 1;
36                 var currentLevel = levels[0];
37                 var previousLevel = levels[0];
38                 var skipOnce = false;
39                 var w = 0;
40                 for (var i = 1; i < length; i++)
41                 {
42                     if (_equalityComparer.Equals(currentLevel, levels[i]))
43                     {
44                         levelRepeat++;
45                         skipOnce = false;
46                         if (levelRepeat == 2)
47                         {
48                             sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
49                             var newLevel = i >= length - 1 ?
50                                 GetPreviousLowerThanCurrentOrCurrent(previousLevel,
                                ↪  currentLevel) :
51                                 i < 2 ?
52                                 GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
```

```
53                              GetGreatestNeigbourLowerThanCurrentOrCurrent(previousLevel,
                            ↪   currentLevel, levels[i + 1]);
54                                  levels[w] = newLevel;
55                                  previousLevel = currentLevel;
56                                  w++;
57                                  levelRepeat = 0;
58                                  skipOnce = true;
59                              }
60                              else if (i == length - 1)
61                              {
62                                  sequence[w] = sequence[i];
63                                  levels[w] = levels[i];
64                                  w++;
65                              }
66                          }
67                          else
68                          {
69                              currentLevel = levels[i];
70                              levelRepeat = 1;
71                              if (skipOnce)
72                              {
73                                  skipOnce = false;
74                              }
75                              else
76                              {
77                                  sequence[w] = sequence[i - 1];
78                                  levels[w] = levels[i - 1];
79                                  previousLevel = levels[w];
80                                  w++;
81                              }
82                              if (i == length - 1)
83                              {
84                                  sequence[w] = sequence[i];
85                                  levels[w] = levels[i];
86                                  w++;
87                              }
88                          }
89                      }
90                      length = w;
91                  }
92              return links.GetOrCreate(sequence[0], sequence[1]);
93          }
94
95          private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink
            ↪   current, TLink next)
96          {
97              return _comparer.Compare(previous, next) > 0
98                  ? _comparer.Compare(previous, current) < 0 ? previous : current
99                  : _comparer.Compare(next, current) < 0 ? next : current;
100         }
101
102         private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
            ↪   _comparer.Compare(next, current) < 0 ? next : current;
103
104         private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
            ↪   => _comparer.Compare(previous, current) < 0 ? previous : current;
105     }
106 }
```

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

```
1   using System.Collections.Generic;
2   using Platform.Interfaces;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.Converters
7   {
8       public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
        ↪   IConverter<IList<TLink>>
9       {
10          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11
12          private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
13
14          public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
            ↪   IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
            ↪   => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
15
16          public IList<TLink> Convert(IList<TLink> sequence)
17          {
```

```
18            var levels = new TLink[sequence.Count];
19            levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
20            for (var i = 1; i < sequence.Count - 1; i++)
21            {
22                var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
23                var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
24                levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
25            }
26            levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
              ↪ sequence[sequence.Count - 1]);
27            return levels;
28        }
29
30        public TLink GetFrequencyNumber(TLink source, TLink target) =>
              ↪ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
31    }
32 }
```

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

```
1  using Platform.Interfaces;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
6  {
7      public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
         ↪ ICriterionMatcher<TLink>
8      {
9          public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
10         public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);
11     }
12 }
```

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

```
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
7  {
8      public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪ EqualityComparer<TLink>.Default;
11
12         private readonly ILinks<TLink> _links;
13         private readonly TLink _sequenceMarkerLink;
14
15         public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
16         {
17             _links = links;
18             _sequenceMarkerLink = sequenceMarkerLink;
19         }
20
21         public bool IsMatched(TLink sequenceCandidate)
22             => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
23             || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
                ↪ sequenceCandidate), _links.Constants.Null);
24     }
25 }
```

./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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences
9  {
10     public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
        ↪ ISequenceAppender<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
13
14         private readonly IStack<TLink> _stack;
```

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

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

```
1 using System.Collections.Generic;
2 using System.Linq;
3 using Platform.Interfaces;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Sequences
8 {
9     public class DuplicateSegmentsCounter<TLink> : ICounter<int>
10    {
11        private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
   ↪   _duplicateFragmentsProvider;
12        public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
   ↪   IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
   ↪   duplicateFragmentsProvider;
13        public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
14    }
15 }
```

./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 using Platform.Data.Doublets.Unicode;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18    public class DuplicateSegmentsProvider<TLink> :
   ↪   DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
   ↪   IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
```

```csharp
    {
        private readonly ILinks<TLink> _links;
        private readonly ISequences<TLink> _sequences;
        private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
        private BitString _visited;

        private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
         ↪ IList<TLink>>>
        {
            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;
        }
```

```csharp
 89          protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
    ↪   length) => new Segment<TLink>(elements, offset, length);
 90
 91          protected override void OnDublicateFound(Segment<TLink> segment)
 92          {
 93              var duplicates = CollectDuplicatesForSegment(segment);
 94              if (duplicates.Count > 1)
 95              {
 96                  _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
    ↪       duplicates));
 97              }
 98          }
 99
100          private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
101          {
102              var duplicates = new List<TLink>();
103              var readAsElement = new HashSet<TLink>();
104              _sequences.Each(sequence =>
105              {
106                  duplicates.Add(sequence);
107                  readAsElement.Add(sequence);
108                  return true; // Continue
109              }, segment);
110              if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
111              {
112                  return new List<TLink>();
113              }
114              foreach (var duplicate in duplicates)
115              {
116                  var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
117                  _visited.Set(duplicateBitIndex);
118              }
119              if (_sequences is Sequences sequencesExperiments)
120              {
121                  var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
    ↪       ashSet<ulong>)(object)readAsElement,
    ↪       (IList<ulong>)segment);
122                  foreach (var partiallyMatchedSequence in partiallyMatched)
123                  {
124                      TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
125                      duplicates.Add(sequenceIndex);
126                  }
127              }
128              duplicates.Sort();
129              return duplicates;
130          }
131
132          private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
133          {
134              if (!(_links is ILinks<ulong> ulongLinks))
135              {
136                  return;
137              }
138              var duplicatesKey = duplicatesItem.Key;
139              var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
140              Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
141              var duplicatesList = duplicatesItem.Value;
142              for (int i = 0; i < duplicatesList.Count; i++)
143              {
144                  ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
145                  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));
146                  Console.WriteLine(formatedSequenceStructure);
147                  var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
    ↪       ulongLinks);
148                  Console.WriteLine(sequenceString);
149              }
150              Console.WriteLine();
151          }
152      }
153  }
```

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

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    /// <remarks>
    /// Can be used to operate with many CompressingConverters (to keep global frequencies data
    /// between them).
    /// TODO: Extract interface to implement frequencies storage inside Links storage
    /// </remarks>
    public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
        private readonly ICounter<TLink, TLink> _frequencyCounter;

        public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
            : base(links)
        {
            _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
                DoubletComparer<TLink>.Default);
            _frequencyCounter = frequencyCounter;
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
        {
            _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
            return data;
        }

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
        {
            if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
            {
                data.IncrementFrequency();
            }
```

```
80                     else
81                     {
82                         var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
83                         data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
84                         if (!_equalityComparer.Equals(link, default))
85                         {
86                             data.Frequency = Arithmetic.Add(data.Frequency,
                             ↪  _frequencyCounter.Count(link));
87                         }
88                         _doubletsCache.Add(doublet, data);
89                     }
90                     return data;
91                 }
92
93             public void ValidateFrequencies()
94             {
95                 foreach (var entry in _doubletsCache)
96                 {
97                     var value = entry.Value;
98                     var linkIndex = value.Link;
99                     if (!_equalityComparer.Equals(linkIndex, default))
100                    {
101                        var frequency = value.Frequency;
102                        var count = _frequencyCounter.Count(linkIndex);
103                        // TODO: Why `frequency` always greater than `count` by 1?
104                        if (((_comparer.Compare(frequency, count) > 0) &&
                            ↪  (_comparer.Compare(Arithmetic.Subtract(frequency, count),
                            ↪  Integer<TLink>.One) > 0))
105                          || ((_comparer.Compare(count, frequency) > 0) &&
                               ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency),
                               ↪  Integer<TLink>.One) > 0)))
106                        {
107                            throw new InvalidOperationException("Frequencies validation failed.");
108                        }
109                    }
110                    //else
111                    //{
112                    //    if (value.Frequency > 0)
113                    //    {
114                    //        var frequency = value.Frequency;
115                    //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
116                    //        var count = _countLinkFrequency(linkIndex);
117
118                    //        if ((frequency > count && frequency - count > 1) || (count > frequency
                           ↪  && count - frequency > 1))
119                    //            throw new Exception("Frequencies validation failed.");
120                    //    }
121                    //}
122                }
123            }
124        }
125    }
```

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

```
1   using System.Runtime.CompilerServices;
2   using Platform.Numbers;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
7   {
8       public class LinkFrequency<TLink>
9       {
10          public TLink Frequency { get; set; }
11          public TLink Link { get; set; }
12
13          public LinkFrequency(TLink frequency, TLink link)
14          {
15              Frequency = frequency;
16              Link = link;
17          }
18
19          public LinkFrequency() { }
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);
```

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

./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.cs
```csharp
1   using Platform.Interfaces;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
6   {
7       public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
        ↪  IConverter<Doublet<TLink>, TLink>
8       {
9           private readonly LinkFrequenciesCache<TLink> _cache;
10          public
            ↪  FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
            ↪  cache) => _cache = cache;
11          public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
12      }
13  }
```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs
```csharp
1   using Platform.Interfaces;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6   {
7       public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
        ↪  SequenceSymbolFrequencyOneOffCounter<TLink>
8       {
9           private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
10
11          public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
            ↪  ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
12              : base(links, sequenceLink, symbol)
13              => _markedSequenceMatcher = markedSequenceMatcher;
14
15          public override TLink Count()
16          {
17              if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
18              {
19                  return default;
20              }
21              return base.Count();
22          }
23      }
24  }
```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs
```csharp
1   using System.Collections.Generic;
2   using Platform.Interfaces;
3   using Platform.Numbers;
4   using Platform.Data.Sequences;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
9   {
10      public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
13          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
14
15          protected readonly ILinks<TLink> _links;
16          protected readonly TLink _sequenceLink;
17          protected readonly TLink _symbol;
18          protected TLink _total;
19
20          public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
            ↪  TLink symbol)
21          {
22              _links = links;
23              _sequenceLink = sequenceLink;
24              _symbol = symbol;
25              _total = default;
26          }
```

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

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

```
1   using Platform.Interfaces;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6   {
7       public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
8       {
9           private readonly ILinks<TLink> _links;
10          private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
11
12          public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
        ↪  ICriterionMatcher<TLink> markedSequenceMatcher)
13          {
14              _links = links;
15              _markedSequenceMatcher = markedSequenceMatcher;
16          }
17
18          public TLink Count(TLink argument) => new
        ↪  TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
        ↪  _markedSequenceMatcher, argument).Count();
19      }
20  }
```

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

```
1   using Platform.Interfaces;
2   using Platform.Numbers;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7   {
8       public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
        ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>
9       {
10          private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
11
12          public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
        ↪  ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
13              : base(links, symbol)
14              => _markedSequenceMatcher = markedSequenceMatcher;
15
16          protected override void CountSequenceSymbolFrequency(TLink link)
17          {
18              var symbolFrequencyCounter = new
        ↪  MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
        ↪  _markedSequenceMatcher, link, _symbol);
19              _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
20          }
21      }
22  }
```

```
1  using Platform.Interfaces;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6  {
7      public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
8      {
9          private readonly ILinks<TLink> _links;
10         public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
11         public TLink Count(TLink symbol) => new
           ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
12     }
13 }
```

```
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
8  {
9      public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
12         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13
14         protected readonly ILinks<TLink> _links;
15         protected readonly TLink _symbol;
16         protected readonly HashSet<TLink> _visits;
17         protected TLink _total;
18
19         public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
20         {
21             _links = links;
22             _symbol = symbol;
23             _visits = new HashSet<TLink>();
24             _total = default;
25         }
26
27         public TLink Count()
28         {
29             if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
30             {
31                 return _total;
32             }
33             CountCore(_symbol);
34             return _total;
35         }
36
37         private void CountCore(TLink link)
38         {
39             var any = _links.Constants.Any;
40             if (_equalityComparer.Equals(_links.Count(any, link), default))
41             {
42                 CountSequenceSymbolFrequency(link);
43             }
44             else
45             {
46                 _links.Each(EachElementHandler, any, link);
47             }
48         }
49
50         protected virtual void CountSequenceSymbolFrequency(TLink link)
51         {
52             var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
               ↪  link, _symbol);
53             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
54         }
55
56         private TLink EachElementHandler(IList<TLink> doublet)
57         {
58             var constants = _links.Constants;
59             var doubletIndex = doublet[constants.IndexPart];
60             if (_visits.Add(doubletIndex))
61             {
```

```
62              CountCore(doubletIndex);
63            }
64            return constants.Continue;
65        }
66    }
67 }
```

./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs
```
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.HeightProviders
7  {
8      public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
         ↪  ISequenceHeightProvider<TLink>
9      {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
11
12          private readonly TLink _heightPropertyMarker;
13          private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
14          private readonly IConverter<TLink> _addressToUnaryNumberConverter;
15          private readonly IConverter<TLink> _unaryNumberToAddressConverter;
16          private readonly IPropertiesOperator<TLink, TLink, TLink> _propertyOperator;
17
18          public CachedSequenceHeightProvider(
19              ILinks<TLink> links,
20              ISequenceHeightProvider<TLink> baseHeightProvider,
21              IConverter<TLink> addressToUnaryNumberConverter,
22              IConverter<TLink> unaryNumberToAddressConverter,
23              TLink heightPropertyMarker,
24              IPropertiesOperator<TLink, TLink, TLink> propertyOperator)
25              : base(links)
26          {
27              _heightPropertyMarker = heightPropertyMarker;
28              _baseHeightProvider = baseHeightProvider;
29              _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
30              _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
31              _propertyOperator = propertyOperator;
32          }
33
34          public TLink Get(TLink sequence)
35          {
36              TLink height;
37              var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
38              if (_equalityComparer.Equals(heightValue, default))
39              {
40                  height = _baseHeightProvider.Get(sequence);
41                  heightValue = _addressToUnaryNumberConverter.Convert(height);
42                  _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
43              }
44              else
45              {
46                  height = _unaryNumberToAddressConverter.Convert(heightValue);
47              }
48              return height;
49          }
50      }
51 }
```

./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs
```
1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.HeightProviders
7  {
8      public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
         ↪  ISequenceHeightProvider<TLink>
9      {
10          private readonly ICriterionMatcher<TLink> _elementMatcher;
11
12          public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
            ↪  elementMatcher) : base(links) => _elementMatcher = elementMatcher;
13
14          public TLink Get(TLink sequence)
15          {
```

```
16        var height = default(TLink);
17        var pairOrElement = sequence;
18        while (!_elementMatcher.IsMatched(pairOrElement))
19        {
20            pairOrElement = Links.GetTarget(pairOrElement);
21            height = Arithmetic.Increment(height);
22        }
23        return height;
24      }
25    }
26  }
```

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

```
1   using Platform.Interfaces;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Sequences.HeightProviders
6   {
7       public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
8       {
9       }
10  }
```

./Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs

```
1   using System.Collections.Generic;
2   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.Indexes
7   {
8       public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
9       {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
11
12          private readonly LinkFrequenciesCache<TLink> _cache;
13
14          public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
            ↪  _cache = cache;
15
16          public bool Add(IList<TLink> sequence)
17          {
18              var indexed = true;
19              var i = sequence.Count;
20              while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
                ↪  { }
21              for (; i >= 1; i--)
22              {
23                  _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
24              }
25              return indexed;
26          }
27
28          private bool IsIndexedWithIncrement(TLink source, TLink target)
29          {
30              var frequency = _cache.GetFrequency(source, target);
31              if (frequency == null)
32              {
33                  return false;
34              }
35              var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
36              if (indexed)
37              {
38                  _cache.IncrementFrequency(source, target);
39              }
40              return indexed;
41          }
42
43          public bool MightContain(IList<TLink> sequence)
44          {
45              var indexed = true;
46              var i = sequence.Count;
47              while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
48              return indexed;
49          }
50
51          private bool IsIndexed(TLink source, TLink target)
52          {
```

```
53          var frequency = _cache.GetFrequency(source, target);
54          if (frequency == null)
55          {
56              return false;
57          }
58          return !_equalityComparer.Equals(frequency.Frequency, default);
59      }
60  }
61 }
```

./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs
```
1  using Platform.Interfaces;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Indexes
7  {
8      public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
       ↪  ISequenceIndex<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
       ↪  EqualityComparer<TLink>.Default;
11
12         private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
13         private readonly IIncrementer<TLink> _frequencyIncrementer;
14
15         public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IPropertyOperator<TLink,
       ↪  TLink> frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
16             : base(links)
17         {
18             _frequencyPropertyOperator = frequencyPropertyOperator;
19             _frequencyIncrementer = frequencyIncrementer;
20         }
21
22         public override bool Add(IList<TLink> sequence)
23         {
24             var indexed = true;
25             var i = sequence.Count;
26             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
       ↪  { }
27             for (; i >= 1; i--)
28             {
29                 Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
30             }
31             return indexed;
32         }
33
34         private bool IsIndexedWithIncrement(TLink source, TLink target)
35         {
36             var link = Links.SearchOrDefault(source, target);
37             var indexed = !_equalityComparer.Equals(link, default);
38             if (indexed)
39             {
40                 Increment(link);
41             }
42             return indexed;
43         }
44
45         private void Increment(TLink link)
46         {
47             var previousFrequency = _frequencyPropertyOperator.Get(link);
48             var frequency = _frequencyIncrementer.Increment(previousFrequency);
49             _frequencyPropertyOperator.Set(link, frequency);
50         }
51     }
52 }
```

./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs
```
1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Indexes
6  {
7      public interface ISequenceIndex<TLink>
8      {
9          /// <summary>
10         /// Индексирует последовательность глобально, и возвращает значение,
```

```
11          /// определяющие была ли запрошенная последовательность проиндексирована ранее.
12          /// </summary>
13          /// <param name="sequence">Последовательность для индексации.</param>
14          bool Add(IList<TLink> sequence);
15
16          bool MightContain(IList<TLink> sequence);
17      }
18  }
```

./Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs
```
1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Indexes
6  {
7      public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
10
11         public SequenceIndex(ILinks<TLink> links) : base(links) { }
12
13         public virtual bool Add(IList<TLink> sequence)
14         {
15             var indexed = true;
16             var i = sequence.Count;
17             while (--i >= 1 && (indexed =
               ↪  !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
               ↪  default))) { }
18             for (; i >= 1; i--)
19             {
20                 Links.GetOrCreate(sequence[i - 1], sequence[i]);
21             }
22             return indexed;
23         }
24
25         public virtual bool MightContain(IList<TLink> sequence)
26         {
27             var indexed = true;
28             var i = sequence.Count;
29             while (--i >= 1 && (indexed =
               ↪  !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
               ↪  default))) { }
30             return indexed;
31         }
32     }
33  }
```

./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs
```
1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Indexes
6  {
7      public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
10
11         private readonly ISynchronizedLinks<TLink> _links;
12
13         public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
14
15         public bool Add(IList<TLink> sequence)
16         {
17             var indexed = true;
18             var i = sequence.Count;
19             var links = _links.Unsync;
20             _links.SyncRoot.ExecuteReadOperation(() =>
21             {
22                 while (--i >= 1 && (indexed =
                   ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                   ↪  sequence[i]), default))) { }
23             });
24             if (!indexed)
25             {
26                 _links.SyncRoot.ExecuteWriteOperation(() =>
```

```
27              {
28                  for (; i >= 1; i--)
29                  {
30                      links.GetOrCreate(sequence[i - 1], sequence[i]);
31                  }
32              });
33          }
34          return indexed;
35      }
36
37      public bool MightContain(IList<TLink> sequence)
38      {
39          var links = _links.Unsync;
40          return _links.SyncRoot.ExecuteReadOperation(() =>
41          {
42              var indexed = true;
43              var i = sequence.Count;
44              while (--i >= 1 && (indexed =
                      !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                      sequence[i]), default))) { }
45              return indexed;
46          });
47      }
48  }
49 }
```

./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 using Platform.Collections.Stacks;
14
15 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17 namespace Platform.Data.Doublets.Sequences
18 {
19     /// <summary>
20     /// Представляет коллекцию последовательностей связей.
21     /// </summary>
22     /// <remarks>
23     /// Обязательно реализовать атомарность каждого публичного метода.
24     ///
25     /// TODO:
26     ///
27     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
28     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
                вместе, все числа вместе и т.п.
29     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
                графа)
30     ///
31     /// x*y - найти все связи между, в последовательностях любой формы, если не стоит
                ограничитель на то, что является последовательностью, а что нет,
32     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
                порядке.
33     ///
34     /// Рост последовательности слева и справа.
35     /// Поиск со звёздочкой.
36     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
37     /// так же проблема может быть решена при реализации дистанционных триггеров.
38     /// Нужны ли уникальные указатели вообще?
39     /// Что если обращение к информации будет происходить через содержимое всегда?
40     ///
41     /// Писать тесты.
42     ///
43     ///
44     /// Можно убрать зависимость от конкретной реализации Links,
45     /// на зависимость от абстрактного элемента, который может быть представлен несколькими
                способами.
46     ///
47     /// Можно ли как-то сделать один общий интерфейс
48     ///
```

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

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

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

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

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

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

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

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

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

            #region Count

            public ulong Count(params ulong[] sequence)
            {
                if (sequence.Length == 0)
                {
                    return Links.Count(_constants.Any, Options.SequenceMarkerLink, _constants.Any);
                }
```

```csharp
            if (sequence.Length == 1) // Первая связь это адрес
            {
                if (sequence[0] == _constants.Null)
                {
                    return 0;
                }
                if (sequence[0] == _constants.Any)
                {
                    return Count();
                }
                if (Options.UseSequenceMarker)
                {
                    return Links.Count(_constants.Any, Options.SequenceMarkerLink, sequence[0]);
                }
                return Links.Exists(sequence[0]) ? 1UL : 0;
            }
            throw new NotImplementedException();
        }

        private ulong CountUsages(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.Index.Add(sequence);
            }
            var sequenceRoot = default(ulong);
            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
            {
                var matches = Each(sequence);
                if (matches.Count > 0)
                {
                    sequenceRoot = matches[0];
                }
            }
            else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
```

```csharp
205            {
206                return CompactCore(sequence);
207            }
208            if (sequenceRoot == default)
209            {
210                sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
211            }
212            if (Options.UseSequenceMarker)
213            {
214                Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
215            }
216            return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
217        }

218
219        #endregion

220
221        #region Each

222
223        public List<ulong> Each(params ulong[] sequence)
224        {
225            var results = new List<ulong>();
226            Each(results.AddAndReturnTrue, sequence);
227            return results;
228        }

229
230        public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
231        {
232            return Sync.ExecuteReadOperation(() =>
233            {
234                if (sequence.IsNullOrEmpty())
235                {
236                    return true;
237                }
238                Links.EnsureEachLinkIsAnyOrExists(sequence);
239                if (sequence.Count == 1)
240                {
241                    var link = sequence[0];
242                    if (link == _constants.Any)
243                    {
244                        return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
245                    }
246                    return handler(link);
247                }
248                if (sequence.Count == 2)
249                {
250                    return Links.Unsync.Each(sequence[0], sequence[1], handler);
251                }
252                if (Options.UseIndex && !Options.Index.MightContain(sequence))
253                {
254                    return false;
255                }
256                return EachCore(handler, sequence);
257            });
258        }

259
260        private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
261        {
262            var matcher = new Matcher(this, sequence, new HashSet<LinkIndex>(), handler);
263            // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
                ↪ Id.
264            Func<ulong, bool> innerHandler = Options.UseSequenceMarker ? (Func<ulong,
                ↪ bool>)matcher.HandleFullMatchedSequence : matcher.HandleFullMatched;
265            //if (sequence.Length >= 2)
266            if (!StepRight(innerHandler, sequence[0], sequence[1]))
267            {
268                return false;
269            }
270            var last = sequence.Count - 2;
271            for (var i = 1; i < last; i++)
272            {
273                if (!PartialStepRight(innerHandler, sequence[i], sequence[i + 1]))
274                {
275                    return false;
276                }
277            }
278            if (sequence.Count >= 3)
279            {
280                if (!StepLeft(innerHandler, sequence[sequence.Count - 2],
                    ↪ sequence[sequence.Count - 1]))
```

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

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

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

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

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

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

        #endregion

        #region Update

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

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

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

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

```
                    {
                        Links.Unsync.MergeUsages(sequence, newSequence);
                    }
                }
            }
        }

        #endregion

        #region Delete

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

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

```csharp
                return Sync.ExecuteWriteOperation(() =>
                {
                    if (sequence.IsNullOrEmpty())
                    {
                        return _constants.Null;
                    }
                    Links.EnsureEachLinkExists(sequence);
                    return CompactCore(sequence);
                });
            }

            [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;
                    foreach (var part in Options.Walker.Walk(sequence))
                    {
                        if (!handler(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, new DefaultStack<ulong>())
                {
                    _sequences = sequences;
                    _patternSequence = patternSequence;
                    _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪  _constants.Any && x != ZeroOrMany));
                    _results = results;
                    _stopableHandler = stopableHandler;
                    _readAsElements = readAsElements;
                }
```

```csharp
            protected override bool IsElement(ulong link) => base.IsElement(link) ||
            ↪  (_readAsElements != null && _readAsElements.Contains(link)) ||
            ↪  _linksInSequence.Contains(link);

            public bool FullMatch(LinkIndex sequenceToMatch)
            {
                _filterPosition = 0;
                foreach (var part in Walk(sequenceToMatch))
                {
                    if (!FullMatchCore(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(part))
                    {
                        break;
                    }
                }
                return _filterPosition == _patternSequence.Count - 1;
            }
```

```
661            private bool PartialMatchCore(LinkIndex element)
662            {
663                if (_filterPosition == (_patternSequence.Count - 1))
664                {
665                    return false; // Нашлось
666                }
667                if (_filterPosition >= 0)
668                {
669                    if (element == _patternSequence[_filterPosition + 1])
670                    {
671                        _filterPosition++;
672                    }
673                    else
674                    {
675                        _filterPosition = -1;
676                    }
677                }
678                if (_filterPosition < 0)
679                {
680                    if (element == _patternSequence[0])
681                    {
682                        _filterPosition = 0;
683                    }
684                }
685                return true; // Ищем дальше
686            }
687
688            public void AddPartialMatchedToResults(ulong sequenceToMatch)
689            {
690                if (PartialMatch(sequenceToMatch))
691                {
692                    _results.Add(sequenceToMatch);
693                }
694            }
695
696            public bool HandlePartialMatched(ulong sequenceToMatch)
697            {
698                if (PartialMatch(sequenceToMatch))
699                {
700                    return _stopableHandler(sequenceToMatch);
701                }
702                return true;
703            }
704
705            public void AddAllPartialMatchedToResults(IEnumerable<ulong> sequencesToMatch)
706            {
707                foreach (var sequenceToMatch in sequencesToMatch)
708                {
709                    if (PartialMatch(sequenceToMatch))
710                    {
711                        _results.Add(sequenceToMatch);
712                    }
713                }
714            }
715
716            public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<ulong>
     ↪    sequencesToMatch)
717            {
718                foreach (var sequenceToMatch in sequencesToMatch)
719                {
720                    if (PartialMatch(sequenceToMatch))
721                    {
722                        _readAsElements.Add(sequenceToMatch);
723                        _results.Add(sequenceToMatch);
724                    }
725                }
726            }
727        }
728
729        #endregion
730    }
731 }
```

./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs
```
1 using System;
2 using LinkIndex = System.UInt64;
3 using System.Collections.Generic;
4 using Stack = System.Collections.Generic.Stack<ulong>;
5 using System.Linq;
```

```csharp
using System.Text;
using Platform.Collections;
using Platform.Data.Exceptions;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Collections.Stacks;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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)Platform.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)
```

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

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

            #endregion

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

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

```csharp
157                     {
158                         var innerSequenceLength = sequence.Length - 1;
159                         for (var li = 0; li < innerSequenceLength; li++)
160                         {
161                             var left = sequence[li];
162                             var right = sequence[li + 1];
163                             if (left == 0 && right == 0)
164                             {
165                                 continue;
166                             }
167                             var linkIndex = li;
168                             ulong[] innerSequence = null;
169                             Links.Unsync.Each(left, right, doublet =>
170                             {
171                                 if (innerSequence == null)
172                                 {
173                                     innerSequence = new ulong[innerSequenceLength];
174                                     for (var isi = 0; isi < linkIndex; isi++)
175                                     {
176                                         innerSequence[isi] = sequence[isi];
177                                     }
178                                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
179                                     {
180                                         innerSequence[isi] = sequence[isi + 1];
181                                     }
182                                 }
183                                 innerSequence[linkIndex] = doublet;
184                                 Each1(handler, innerSequence);
185                                 return _constants.Continue;
186                             });
187                         }
188                     }
189             }
190
191             public HashSet<ulong> EachPart(params ulong[] sequence)
192             {
193                 var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
194                 EachPartCore(link =>
195                 {
196                     if (!visitedLinks.Contains(link))
197                     {
198                         visitedLinks.Add(link); // изучить почему случаются повторы
199                     }
200                     return true;
201                 }, sequence);
202                 return visitedLinks;
203             }
204
205             public void EachPart(Func<ulong, bool> handler, params ulong[] sequence)
206             {
207                 var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
208                 EachPartCore(link =>
209                 {
210                     if (!visitedLinks.Contains(link))
211                     {
212                         visitedLinks.Add(link); // изучить почему случаются повторы
213                         return handler(link);
214                     }
215                     return true;
216                 }, sequence);
217             }
218
219             private void EachPartCore(Func<ulong, bool> handler, params ulong[] sequence)
220             {
221                 if (sequence.IsNullOrEmpty())
222                 {
223                     return;
224                 }
225                 Links.EnsureEachLinkIsAnyOrExists(sequence);
226                 if (sequence.Length == 1)
227                 {
228                     var link = sequence[0];
229                     if (link > 0)
230                     {
231                         handler(link);
232                     }
233                     else
234                     {
235                         Links.Each(_constants.Any, _constants.Any, handler);
```

```csharp
                }
            }
            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);
                    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 =>
```

```
315                    {
316                        StepLeft(handler, left, doublet);
317                        if (right != doublet)
318                        {
319                            PartialStepLeft(handler, left, doublet);
320                        }
321                        return true;
322                    });
323            }
324
325            private void StepLeft(Action<ulong> handler, ulong left, ulong right)
326            {
327                Links.Unsync.Each(_constants.Any, right, leftStep =>
328                {
329                    TryStepLeftUp(handler, left, leftStep);
330                    return true;
331                });
332            }
333
334            private void TryStepLeftUp(Action<ulong> handler, ulong left, ulong stepFrom)
335            {
336                var upStep = stepFrom;
337                var firstTarget = Links.Unsync.GetSource(upStep);
338                while (firstTarget != left && firstTarget != upStep)
339                {
340                    upStep = firstTarget;
341                    firstTarget = Links.Unsync.GetTarget(upStep);
342                }
343                if (firstTarget == left)
344                {
345                    handler(stepFrom);
346                }
347            }
348
349            private bool StartsWith(ulong sequence, ulong link)
350            {
351                var upStep = sequence;
352                var firstSource = Links.Unsync.GetSource(upStep);
353                while (firstSource != link && firstSource != upStep)
354                {
355                    upStep = firstSource;
356                    firstSource = Links.Unsync.GetSource(upStep);
357                }
358                return firstSource == link;
359            }
360
361            private bool EndsWith(ulong sequence, ulong link)
362            {
363                var upStep = sequence;
364                var lastTarget = Links.Unsync.GetTarget(upStep);
365                while (lastTarget != link && lastTarget != upStep)
366                {
367                    upStep = lastTarget;
368                    lastTarget = Links.Unsync.GetTarget(upStep);
369                }
370                return lastTarget == link;
371            }
372
373            public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
374            {
375                return Sync.ExecuteReadOperation(() =>
376                {
377                    var results = new List<ulong>();
378                    if (sequence.Length > 0)
379                    {
380                        Links.EnsureEachLinkExists(sequence);
381                        var firstElement = sequence[0];
382                        if (sequence.Length == 1)
383                        {
384                            results.Add(firstElement);
385                            return results;
386                        }
387                        if (sequence.Length == 2)
388                        {
389                            var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
390                            if (doublet != _constants.Null)
391                            {
392                                results.Add(doublet);
393                            }
```

```
394                                    return results;
395                                }
396                            var linksInSequence = new HashSet<ulong>(sequence);
397                            void handler(ulong result)
398                            {
399                                var filterPosition = 0;
400                                StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
                                 ↪  Links.Unsync.GetTarget,
401                                    x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                                 ↪  x =>
402                                    {
403                                        if (filterPosition == sequence.Length)
404                                        {
405                                            filterPosition = -2; // Длиннее чем нужно
406                                            return false;
407                                        }
408                                        if (x != sequence[filterPosition])
409                                        {
410                                            filterPosition = -1;
411                                            return false; // Начинается иначе
412                                        }
413                                        filterPosition++;

415                                        return true;
416                                    });
417                                if (filterPosition == sequence.Length)
418                                {
419                                    results.Add(result);
420                                }
421                            }
422                            if (sequence.Length >= 2)
423                            {
424                                StepRight(handler, sequence[0], sequence[1]);
425                            }
426                            var last = sequence.Length - 2;
427                            for (var i = 1; i < last; i++)
428                            {
429                                PartialStepRight(handler, sequence[i], sequence[i + 1]);
430                            }
431                            if (sequence.Length >= 3)
432                            {
433                                StepLeft(handler, sequence[sequence.Length - 2],
                                 ↪  sequence[sequence.Length - 1]);
434                            }
435                        }
436                        return results;
437                    });
438            }

440            public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
441            {
442                return Sync.ExecuteReadOperation(() =>
443                {
444                    var results = new HashSet<ulong>();
445                    if (sequence.Length > 0)
446                    {
447                        Links.EnsureEachLinkExists(sequence);
448                        var firstElement = sequence[0];
449                        if (sequence.Length == 1)
450                        {
451                            results.Add(firstElement);
452                            return results;
453                        }
454                        if (sequence.Length == 2)
455                        {
456                            var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
457                            if (doublet != _constants.Null)
458                            {
459                                results.Add(doublet);
460                            }
461                            return results;
462                        }
463                        var matcher = new Matcher(this, sequence, results, null);
464                        if (sequence.Length >= 2)
465                        {
466                            StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
467                        }
468                        var last = sequence.Length - 2;
469                        for (var i = 1; i < last; i++)
```

```csharp
                    {
                        PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
                        ↪   sequence[i + 1]);
                    }
                    if (sequence.Length >= 3)
                    {
                        StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
                        ↪   sequence[sequence.Length - 1]);
                    }
                }
                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>();
```

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

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

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

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

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

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

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

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

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

    //            firstResults.IntersectWith(lastResults);

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

```
684     //              //     AllUsagesCore(sequence[i], results);

685

686     //                  var filteredResults = new HashSet<ulong>();
687     //                  var matcher = new Matcher(this, sequence, filteredResults, null);
688     //                  matcher.AddAllPartialMatchedToResults(firstResults);
689     //                  return filteredResults;
690     //              }

691

692     //          return new HashSet<ulong>();
693     //      });
694     //}

695

696     public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
697     {
698         return Sync.ExecuteReadOperation(() =>
699         {
700             if (sequence.Length > 0)
701             {
702                 Links.EnsureEachLinkIsAnyOrExists(sequence);
703                 var firstResults = new HashSet<ulong>();
704                 var lastResults = new HashSet<ulong>();
705                 var first = sequence.First(x => x != _constants.Any);
706                 var last = sequence.Last(x => x != _constants.Any);
707                 AllUsagesCore(first, firstResults);
708                 AllUsagesCore(last, lastResults);
709                 firstResults.IntersectWith(lastResults);
710                 //for (var i = 0; i < sequence.Length; i++)
711                 //     AllUsagesCore(sequence[i], results);
712                 var filteredResults = new HashSet<ulong>();
713                 var matcher = new Matcher(this, sequence, filteredResults, null);
714                 matcher.AddAllPartialMatchedToResults(firstResults);
715                 return filteredResults;
716             }
717             return new HashSet<ulong>();
718         });
719     }

720

721     public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
        ↪  IList<ulong> sequence)
722     {
723         return Sync.ExecuteReadOperation(() =>
724         {
725             if (sequence.Count > 0)
726             {
727                 Links.EnsureEachLinkExists(sequence);
728                 var results = new HashSet<LinkIndex>();
729                 //var nextResults = new HashSet<ulong>();
730                 //for (var i = 0; i < sequence.Length; i++)
731                 //{
732                 //     AllUsagesCore(sequence[i], nextResults);
733                 //     if (results.IsNullOrEmpty())
734                 //     {
735                 //         results = nextResults;
736                 //         nextResults = new HashSet<ulong>();
737                 //     }
738                 //     else
739                 //     {
740                 //         results.IntersectWith(nextResults);
741                 //         nextResults.Clear();
742                 //     }
743                 //}
744                 var collector1 = new AllUsagesCollector1(Links.Unsync, results);
745                 collector1.Collect(Links.Unsync.GetLink(sequence[0]));
746                 var next = new HashSet<ulong>();
747                 for (var i = 1; i < sequence.Count; i++)
748                 {
749                     var collector = new AllUsagesCollector1(Links.Unsync, next);
750                     collector.Collect(Links.Unsync.GetLink(sequence[i]));

751
752                     results.IntersectWith(next);
753                     next.Clear();
754                 }
755                 var filteredResults = new HashSet<ulong>();
756                 var matcher = new Matcher(this, sequence, filteredResults, null,
                    ↪  readAsElements);
757                 matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
                    ↪  x)); // OrderBy is a Hack
758                 return filteredResults;
```

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

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

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

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

/// <remarks>
/// TODO: Может потроботаться 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)
{
```

```
903            return Sync.ExecuteReadOperation(() =>
904            {
905                var visits = new HashSet<ulong>();
906                var usages = new HashSet<ulong>();
907                AllBottomUsagesCore(link, visits, usages);
908                return usages;
909            });
910        }
911
912        private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
           ↪  usages)
913        {
914            bool handler(ulong doublet)
915            {
916                if (visits.Add(doublet))
917                {
918                    AllBottomUsagesCore(doublet, visits, usages);
919                }
920                return true;
921            }
922            if (Links.Unsync.Count(_constants.Any, link) == 0)
923            {
924                usages.Add(link);
925            }
926            else
927            {
928                Links.Unsync.Each(link, _constants.Any, handler);
929                Links.Unsync.Each(_constants.Any, link, handler);
930            }
931        }
932
933        public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
934        {
935            if (Options.UseSequenceMarker)
936            {
937                var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                   ↪  Options.MarkedSequenceMatcher, symbol);
938                return counter.Count();
939            }
940            else
941            {
942                var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                   ↪  symbol);
943                return counter.Count();
944            }
945        }
946
947        private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool>
           ↪  outerHandler)
948        {
949            bool handler(ulong doublet)
950            {
951                if (usages.Add(doublet))
952                {
953                    if (!outerHandler(doublet))
954                    {
955                        return false;
956                    }
957                    if (!AllUsagesCore1(doublet, usages, outerHandler))
958                    {
959                        return false;
960                    }
961                }
962                return true;
963            }
964            return Links.Unsync.Each(link, _constants.Any, handler)
965                && Links.Unsync.Each(_constants.Any, link, handler);
966        }
967
968        public void CalculateAllUsages(ulong[] totals)
969        {
970            var calculator = new AllUsagesCalculator(Links, totals);
971            calculator.Calculate();
972        }
973
974        public void CalculateAllUsages2(ulong[] totals)
975        {
976            var calculator = new AllUsagesCalculator2(Links, totals);
```

```csharp
                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;
            }

            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)
                    {
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
1356                   for (var i = 1; i < linksToConnect.Length; i++)
1357                   {
1358                       var next = new HashSet<ulong>();
1359                       var collector = new AllUsagesIntersectingCollector(Links, results, next);
1360                       collector.Collect(linksToConnect[i]);
1361                       //AllUsagesCore(linksToConnect[i], next);
1362                       //results.IntersectWith(next);
1363                       results = next;
1364                   }
1365               }
1366               return results;
1367           });
1368       }
1369
1370       public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1371       {
1372           return Sync.ExecuteReadOperation(() =>
1373           {
1374               var results = new BitString((long)Links.Unsync.Count() + 1); // new
                   ↪ BitArray((int)_links.Total + 1);
1375               if (linksToConnect.Length > 0)
1376               {
1377                   Links.EnsureEachLinkExists(linksToConnect);
1378                   var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1379                   collector1.Collect(linksToConnect[0]);
1380                   for (var i = 1; i < linksToConnect.Length; i++)
1381                   {
1382                       var next = new BitString((long)Links.Unsync.Count() + 1); //new
                           ↪ BitArray((int)_links.Total + 1);
1383                       var collector = new AllUsagesCollector2(Links.Unsync, next);
1384                       collector.Collect(linksToConnect[i]);
1385                       results = results.And(next);
1386                   }
1387               }
1388               return results.GetSetUInt64Indices();
1389           });
1390       }
1391
1392       private static ulong[] Simplify(ulong[] sequence)
1393       {
1394           // Считаем новый размер последовательности
1395           long newLength = 0;
1396           var zeroOrManyStepped = false;
1397           for (var i = 0; i < sequence.Length; i++)
1398           {
1399               if (sequence[i] == ZeroOrMany)
1400               {
1401                   if (zeroOrManyStepped)
1402                   {
1403                       continue;
1404                   }
1405                   zeroOrManyStepped = true;
1406               }
1407               else
1408               {
1409                   //if (zeroOrManyStepped) Is it efficient?
1410                   zeroOrManyStepped = false;
1411               }
1412               newLength++;
1413           }
1414           // Строим новую последовательность
1415           zeroOrManyStepped = false;
1416           var newSequence = new ulong[newLength];
1417           long j = 0;
1418           for (var i = 0; i < sequence.Length; i++)
1419           {
1420               //var current = zeroOrManyStepped;
1421               //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1422               //if (current && zeroOrManyStepped)
1423               //    continue;
1424               //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1425               //if (zeroOrManyStepped && newZeroOrManyStepped)
1426               //    continue;
1427               //zeroOrManyStepped = newZeroOrManyStepped;
1428               if (sequence[i] == ZeroOrMany)
1429               {
1430                   if (zeroOrManyStepped)
1431                   {
1432                       continue;
```

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

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

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

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

        #region From Triplets

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

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

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

```
1508                            }
1509                        }
1510                    }
1511                }
1512                else
1513                {
1514                    var nextRightLink = middleLinks[rightBound];
1515                    var elements = GetLeftElements(rightLink, nextRightLink);
1516                    if (leftBound <= rightBound)
1517                    {
1518                        for (var i = elements.Length - 1; i >= 0; i--)
1519                        {
1520                            var element = elements[i];
1521                            if (element != 0)
1522                            {
1523                                CollectMatchingSequences(leftLink, leftBound, middleLinks,
                                    ↪  elements[i], rightBound - 1, ref results);
1524                            }
1525                        }
1526                    }
1527                    else
1528                    {
1529                        for (var i = elements.Length - 1; i >= 0; i--)
1530                        {
1531                            var element = elements[i];
1532                            if (element != 0)
1533                            {
1534                                results.Add(element);
1535                            }
1536                        }
1537                    }
1538                }
1539            }
1540
1541            public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1542            {
1543                var result = new ulong[5];
1544                TryStepRight(startLink, rightLink, result, 0);
1545                Links.Each(_constants.Any, startLink, couple =>
1546                {
1547                    if (couple != startLink)
1548                    {
1549                        if (TryStepRight(couple, rightLink, result, 2))
1550                        {
1551                            return false;
1552                        }
1553                    }
1554                    return true;
1555                });
1556                if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1557                {
1558                    result[4] = startLink;
1559                }
1560                return result;
1561            }
1562
1563            public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1564            {
1565                var added = 0;
1566                Links.Each(startLink, _constants.Any, couple =>
1567                {
1568                    if (couple != startLink)
1569                    {
1570                        var coupleTarget = Links.GetTarget(couple);
1571                        if (coupleTarget == rightLink)
1572                        {
1573                            result[offset] = couple;
1574                            if (++added == 2)
1575                            {
1576                                return false;
1577                            }
1578                        }
1579                        else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
                            ↪  == Net.And &&
1580                        {
1581                            result[offset + 1] = couple;
1582                            if (++added == 2)
1583                            {
```

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

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

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

    #endregion

    #region Walkers

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

        #region Pattern Match

        enum PatternBlockType
        {
            Undefined,
            Gap,
            Elements
```

```csharp
            }

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

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

            #endregion

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

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

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

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

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

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

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

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

```csharp
1818        //     {    /* a * matches zero or more instances */
1819        //         if (matchhere(regexp, text))
1820        //             return 1;
1821        //     } while (*text != '\0' && (*text++ == c || c == '.'));
1822        //     return 0;
1823        //}
1824
1825        //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
            ↪ long maximumGap)
1826        //{
1827        //     mininumGap = 0;
1828        //     maximumGap = 0;
1829        //     element = 0;
1830        //     for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1831        //     {
1832        //         if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1833        //             mininumGap++;
1834        //         else if (_patternSequence[_patternPosition] == ZeroOrMany)
1835        //             maximumGap = long.MaxValue;
1836        //         else
1837        //             break;
1838        //     }
1839
1840        //     if (maximumGap < mininumGap)
1841        //         maximumGap = mininumGap;
1842        //}
1843
1844        private bool PatternMatchCore(LinkIndex element)
1845        {
1846            if (_patternPosition >= _pattern.Count)
1847            {
1848                _patternPosition = -2;
1849                return false;
1850            }
1851            var currentPatternBlock = _pattern[_patternPosition];
1852            if (currentPatternBlock.Type == PatternBlockType.Gap)
1853            {
1854                //var currentMatchingBlockLength = (_sequencePosition -
                    ↪ _lastMatchedBlockPosition);
1855                if (_sequencePosition < currentPatternBlock.Start)
1856                {
1857                    _sequencePosition++;
1858                    return true; // Двигаемся дальше
1859                }
1860                // Это последний блок
1861                if (_pattern.Count == _patternPosition + 1)
1862                {
1863                    _patternPosition++;
1864                    _sequencePosition = 0;
1865                    return false; // Полное соответствие
1866                }
1867                else
1868                {
1869                    if (_sequencePosition > currentPatternBlock.Stop)
1870                    {
1871                        return false; // Соответствие невозможно
1872                    }
1873                    var nextPatternBlock = _pattern[_patternPosition + 1];
1874                    if (_patternSequence[nextPatternBlock.Start] == element)
1875                    {
1876                        if (nextPatternBlock.Start < nextPatternBlock.Stop)
1877                        {
1878                            _patternPosition++;
1879                            _sequencePosition = 1;
1880                        }
1881                        else
1882                        {
1883                            _patternPosition += 2;
1884                            _sequencePosition = 0;
1885                        }
1886                    }
1887                }
1888            }
1889            else // currentPatternBlock.Type == PatternBlockType.Elements
1890            {
1891                var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1892                if (_patternSequence[patternElementPosition] != element)
1893                {
1894                    return false; // Соответствие невозможно
```

```
1895                        }
1896                        if (patternElementPosition == currentPatternBlock.Stop)
1897                        {
1898                            _patternPosition++;
1899                            _sequencePosition = 0;
1900                        }
1901                        else
1902                        {
1903                            _sequencePosition++;
1904                        }
1905                    }
1906                    return true;
1907                    //if (_patternSequence[_patternPosition] != element)
1908                    //    return false;
1909                    //else
1910                    //{
1911                    //    _sequencePosition++;
1912                    //    _patternPosition++;
1913                    //    return true;
1914                    //}
1915                    ////////
1916                    //if (_filterPosition == _patternSequence.Length)
1917                    //{
1918                    //    _filterPosition = -2; // Длиннее чем нужно
1919                    //    return false;
1920                    //}
1921                    //if (element != _patternSequence[_filterPosition])
1922                    //{
1923                    //    _filterPosition = -1;
1924                    //    return false; // Начинается иначе
1925                    //}
1926                    //_filterPosition++;
1927                    //if (_filterPosition == (_patternSequence.Length - 1))
1928                    //    return false;
1929                    //if (_filterPosition >= 0)
1930                    //{
1931                    //    if (element == _patternSequence[_filterPosition + 1])
1932                    //        _filterPosition++;
1933                    //    else
1934                    //        return false;
1935                    //}
1936                    //if (_filterPosition < 0)
1937                    //{
1938                    //    if (element == _patternSequence[0])
1939                    //        _filterPosition = 0;
1940                    //}
1941                }

1943            public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
1944            {
1945                foreach (var sequenceToMatch in sequencesToMatch)
1946                {
1947                    if (PatternMatch(sequenceToMatch))
1948                    {
1949                        _results.Add(sequenceToMatch);
1950                    }
1951                }
1952            }
1953        }

1955        #endregion
1956    }
1957 }
```

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

```
1  using Platform.Collections.Lists;
2  using Platform.Data.Sequences;
3  using System.Collections.Generic;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences
8  {
9      public static class SequencesExtensions
10     {
11         public static TLink Create<TLink>(this ISequences<TLink> sequences, IList<TLink[]>
            ↪ groupedSequence)
12         {
13             var finalSequence = new TLink[groupedSequence.Count];
```

```
14                    for (var i = 0; i < finalSequence.Length; i++)
15                    {
16                        var part = groupedSequence[i];
17                        finalSequence[i] = part.Length == 1 ? part[0] : sequences.Create(part);
18                    }
19                    return sequences.Create(finalSequence);
20                }

22                public static IList<TLink> ToList<TLink>(this ISequences<TLink> sequences, TLink
    ↪   sequence)
23                {
24                    var list = new List<TLink>();
25                    sequences.EachPart(list.AddAndReturnTrue, sequence);
26                    return list;
27                }
28            }
29        }
```

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

```
 1   using System;
 2   using System.Collections.Generic;
 3   using Platform.Interfaces;
 4   using Platform.Collections.Stacks;
 5   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
 6   using Platform.Data.Doublets.Sequences.Frequencies.Counters;
 7   using Platform.Data.Doublets.Sequences.Converters;
 8   using Platform.Data.Doublets.Sequences.CreteriaMatchers;
 9   using Platform.Data.Doublets.Sequences.Walkers;
10   using Platform.Data.Doublets.Sequences.Indexes;
11
12   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14   namespace Platform.Data.Doublets.Sequences
15   {
16       public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
         ↪   ILinks<TLink> must contain GetConstants function.
17       {
18           private static readonly EqualityComparer<TLink> _equalityComparer =
             ↪   EqualityComparer<TLink>.Default;
19
20           public TLink SequenceMarkerLink { get; set; }
21           public bool UseCascadeUpdate { get; set; }
22           public bool UseCascadeDelete { get; set; }
23           public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
24           public bool UseSequenceMarker { get; set; }
25           public bool UseCompression { get; set; }
26           public bool UseGarbageCollection { get; set; }
27           public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
28           public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
29
30           public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher { get; set; }
31           public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
32           public ISequenceIndex<TLink> Index { get; set; }
33           public ISequenceWalker<TLink> Walker { get; set; }
34
35           // TODO: Реализовать компактификацию при чтении
36           //public bool EnforceSingleSequenceVersionOnRead { get; set; }
37           //public bool UseRequestMarker { get; set; }
38           //public bool StoreRequestResults { get; set; }
39
40           public void InitOptions(ISynchronizedLinks<TLink> links)
41           {
42               if (UseSequenceMarker)
43               {
44                   if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
45                   {
46                       SequenceMarkerLink = links.CreatePoint();
47                   }
48                   else
49                   {
50                       if (!links.Exists(SequenceMarkerLink))
51                       {
52                           var link = links.CreatePoint();
53                           if (!_equalityComparer.Equals(link, SequenceMarkerLink))
54                           {
55                               throw new InvalidOperationException("Cannot recreate sequence marker
                              ↪   link.");
56                           }
57                       }
```

```
58                  }
59                  if (MarkedSequenceMatcher == null)
60                  {
61                      MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                        ↪   SequenceMarkerLink);
62                  }
63              }
64              var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
65              if (UseCompression)
66              {
67                  if (LinksToSequenceConverter == null)
68                  {
69                      ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
70                      if (UseSequenceMarker)
71                      {
72                          totalSequenceSymbolFrequencyCounter = new
                            ↪   TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                            ↪   MarkedSequenceMatcher);
73                      }
74                      else
75                      {
76                          totalSequenceSymbolFrequencyCounter = new
                            ↪   TotalSequenceSymbolFrequencyCounter<TLink>(links);
77                      }
78                      var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                        ↪   totalSequenceSymbolFrequencyCounter);
79                      var compressingConverter = new CompressingConverter<TLink>(links,
                        ↪   balancedVariantConverter, doubletFrequenciesCache);
80                      LinksToSequenceConverter = compressingConverter;
81                  }
82              }
83              else
84              {
85                  if (LinksToSequenceConverter == null)
86                  {
87                      LinksToSequenceConverter = balancedVariantConverter;
88                  }
89              }
90              if (UseIndex && Index == null)
91              {
92                  Index = new SequenceIndex<TLink>(links);
93              }
94              if (Walker == null)
95              {
96                  Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
97              }
98          }
99
100         public void ValidateOptions()
101         {
102             if (UseGarbageCollection && !UseSequenceMarker)
103             {
104                 throw new NotSupportedException("To use garbage collection UseSequenceMarker
                    ↪   option must be on.");
105             }
106         }
107     }
108 }
```

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

```
1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Walkers
6  {
7      public interface ISequenceWalker<TLink>
8      {
9          IEnumerable<TLink> Walk(TLink sequence);
10     }
11 }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Stacks;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
```

```csharp
namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack)
            { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPop(TLink element) =>
            Links.GetSource(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPush(TLink element) =>
            Links.GetTarget(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLink> WalkContents(TLink element)
        {
            var parts = Links.GetLink(element);
            var start = Links.Constants.IndexPart + 1;
            for (var i = parts.Count - 1; i >= start; i--)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

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

```csharp
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

//#define USEARRAYPOOL
#if USEARRAYPOOL
using Platform.Collections;
#endif

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;

        private readonly Func<TLink, bool> _isElement;

        public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
            base(links) => _isElement = isElement;

        public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
            Links.IsPartialPoint;

        public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);

        public TLink[] ToArray(TLink sequence)
        {
            var length = 1;
            var array = new TLink[length];
            array[0] = sequence;
            if (_isElement(sequence))
            {
                return array;
            }
            bool hasElements;
            do
            {
                length *= 2;
#if USEARRAYPOOL
                var nextArray = ArrayPool.Allocate<ulong>(length);
#else
                var nextArray = new TLink[length];
#endif
```

```csharp
                        hasElements = false;
                        for (var i = 0; i < array.Length; i++)
                        {
                            var candidate = array[i];
                            if (_equalityComparer.Equals(array[i], default))
                            {
                                continue;
                            }
                            var doubletOffset = i * 2;
                            if (_isElement(candidate))
                            {
                                nextArray[doubletOffset] = candidate;
                            }
                            else
                            {
                                var link = Links.GetLink(candidate);
                                var linkSource = Links.GetSource(link);
                                var linkTarget = Links.GetTarget(link);
                                nextArray[doubletOffset] = linkSource;
                                nextArray[doubletOffset + 1] = linkTarget;
                                if (!hasElements)
                                {
                                    hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
                                }
                            }
                        }
#if USEARRAYPOOL
                        if (array.Length > 1)
                        {
                            ArrayPool.Free(array);
                        }
#endif
                        array = nextArray;
                    }
                    while (hasElements);
                    var filledElementsCount = CountFilledElements(array);
                    if (filledElementsCount == array.Length)
                    {
                        return array;
                    }
                    else
                    {
                        return CopyFilledElements(array, filledElementsCount);
                    }
                }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static int CountFilledElements(TLink[] array)
        {
            var count = 0;
            for (var i = 0; i < array.Length; i++)
            {
                if (!_equalityComparer.Equals(array[i], default))
                {
                    count++;
                }
            }
            return count;
        }
    }
}
```

```csharp
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
          stack) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPop(TLink element) =>
          Links.GetTarget(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPush(TLink element) =>
          Links.GetSource(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLink> WalkContents(TLink element)
        {
            var parts = Links.GetLink(element);
            for (var i = Links.Constants.IndexPart + 1; i < parts.Count; i++)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

```csharp
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
      ISequenceWalker<TLink>
    {
        private readonly IStack<TLink> _stack;

        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : base(links) =>
          _stack = stack;

        public IEnumerable<TLink> Walk(TLink sequence)
        {
            _stack.Clear();
            var element = sequence;
            if (IsElement(element))
            {
                yield return element;
            }
            else
            {
                while (true)
                {
                    if (IsElement(element))
                    {
                        if (_stack.IsEmpty)
                        {
                            break;
                        }
                        element = _stack.Pop();
                        foreach (var output in WalkContents(element))
                        {
                            yield return output;
                        }
                        element = GetNextElementAfterPop(element);
```

```
39                         }
40                         else
41                         {
42                             _stack.Push(element);
43                             element = GetNextElementAfterPush(element);
44                         }
45                     }
46                 }
47             }
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected virtual bool IsElement(TLink elementLink) => Links.IsPartialPoint(elementLink);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected abstract TLink GetNextElementAfterPop(TLink element);
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected abstract TLink GetNextElementAfterPush(TLink element);
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected abstract IEnumerable<TLink> WalkContents(TLink element);
60     }
61 }
```

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

```
1  using System.Collections.Generic;
2  using Platform.Collections.Stacks;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Stacks
7  {
8      public class Stack<TLink> : IStack<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
11
12         private readonly ILinks<TLink> _links;
13         private readonly TLink _stack;
14
15         public bool IsEmpty => _equalityComparer.Equals(Peek(), _stack);
16
17         public Stack(ILinks<TLink> links, TLink stack)
18         {
19             _links = links;
20             _stack = stack;
21         }
22
23         private TLink GetStackMarker() => _links.GetSource(_stack);
24
25         private TLink GetTop() => _links.GetTarget(_stack);
26
27         public TLink Peek() => _links.GetTarget(GetTop());
28
29         public TLink Pop()
30         {
31             var element = Peek();
32             if (!_equalityComparer.Equals(element, _stack))
33             {
34                 var top = GetTop();
35                 var previousTop = _links.GetSource(top);
36                 _links.Update(_stack, GetStackMarker(), previousTop);
37                 _links.Delete(top);
38             }
39             return element;
40         }
41
42         public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
            ↪  _links.GetOrCreate(GetTop(), element));
43     }
44 }
```

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

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets.Stacks
4  {
5      public static class StackExtensions
6      {
7          public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
```

```
 8            {
 9                var stackPoint = links.CreatePoint();
10                var stack = links.Update(stackPoint, stackMarker, stackPoint);
11                return stack;
12            }
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    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 8
 9    namespace Platform.Data.Doublets
10    {
11        /// <remarks>
12        /// TODO: Autogeneration of synchronized wrapper (decorator).
13        /// TODO: Try to unfold code of each method using IL generation for performance improvements.
14        /// TODO: Or even to unfold multiple layers of implementations.
15        /// </remarks>
16        public class SynchronizedLinks<T> : ISynchronizedLinks<T>
17        {
18            public LinksCombinedConstants<T, T, int> Constants { get; }
19            public ISynchronization SyncRoot { get; }
20            public ILinks<T> Sync { get; }
21            public ILinks<T> Unsync { get; }
22
23            public SynchronizedLinks(ILinks<T> links) : this(new ReaderWriterLockSynchronization(),
     ↪   links) { }
24
25            public SynchronizedLinks(ISynchronization synchronization, ILinks<T> links)
26            {
27                SyncRoot = synchronization;
28                Sync = this;
29                Unsync = links;
30                Constants = links.Constants;
31            }
32
33            public T Count(IList<T> restriction) => SyncRoot.ExecuteReadOperation(restriction,
     ↪   Unsync.Count);
34            public T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
     ↪   SyncRoot.ExecuteReadOperation(handler, restrictions, (handler1, restrictions1) =>
     ↪   Unsync.Each(handler1, restrictions1));
35            public T Create() => SyncRoot.ExecuteWriteOperation(Unsync.Create);
36            public T Update(IList<T> restrictions) => SyncRoot.ExecuteWriteOperation(restrictions,
     ↪   Unsync.Update);
37            public void Delete(T link) => SyncRoot.ExecuteWriteOperation(link, Unsync.Delete);
38
39            //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
     ↪   IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
40            //{
41            //    if (restriction != null && substitution != null &&
     ↪   !substitution.EqualTo(restriction))
42            //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
     ↪   substitution, substitutedHandler, Unsync.Trigger);
43
44            //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
     ↪   substitutedHandler, Unsync.Trigger);
45            //}
46        }
47    }
```

./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    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12    namespace Platform.Data.Doublets
```

```csharp
{
    /// <summary>
    /// Структура описывающая уникальную связь.
    /// </summary>
    public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
    {
        private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
            Default<LinksCombinedConstants<bool, ulong, int>>.Instance;

        private const int Length = 3;

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

        public static readonly UInt64Link Null = new UInt64Link();

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

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

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

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

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

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

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

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

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

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

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

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

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

        public 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;
        }
        if (index == _constants.TargetPart)
        {
            return Target;
        }
        throw new NotSupportedException(); // Impossible path due to
            Ensure.ArgumentInRange
    }
    set => throw new NotSupportedException();
}

public int Count => Length;

public bool IsReadOnly => true;

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

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

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

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

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

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

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

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

    return -1;
}

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

```
158
159        public void RemoveAt(int index) => throw new NotSupportedException();
160
161        #endregion
162    }
163 }
```

./Platform.Data.Doublets/UInt64LinkExtensions.cs
```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets
4  {
5      public static class UInt64LinkExtensions
6      {
7          public static bool IsFullPoint(this UInt64Link link) => Point<ulong>.IsFullPoint(link);
8          public static bool IsPartialPoint(this UInt64Link link) =>
            ↪  Point<ulong>.IsPartialPoint(link);
9      }
10 }
```

./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.Unicode;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets
12 {
13     public static class UInt64LinksExtensions
14     {
15         public static readonly LinksCombinedConstants<bool, ulong, int> Constants =
           ↪  Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
16
17         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
18
19         public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
20         {
21             if (sequence == null)
22             {
23                 return;
24             }
25             for (var i = 0; i < sequence.Count; i++)
26             {
27                 if (!links.Exists(sequence[i]))
28                 {
29                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                       ↪  $"sequence[{i}]");
30                 }
31             }
32         }
33
34         public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
           ↪  sequence)
35         {
36             if (sequence == null)
37             {
38                 return;
39             }
40             for (var i = 0; i < sequence.Count; i++)
41             {
42                 if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
43                 {
44                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                       ↪  $"sequence[{i}]");
45                 }
46             }
47         }
48
49         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
50         {
51             if (sequence == null)
52             {
53                 return false;
54             }
```

```csharp
                    var constants = links.Constants;
                    for (var i = 0; i < sequence.Length; i++)
                    {
                        if (sequence[i] == constants.Any)
                        {
                            return true;
                        }
                    }
                    return false;
                }

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

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

                public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
                ↪  HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
                ↪  Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false, bool
                ↪  renderDebug = false)
                {
                    if (sb == null)
                    {
                        throw new ArgumentNullException(nameof(sb));
                    }
                    if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
                    ↪  Constants.Itself)
                    {
                        return;
                    }
                    if (links.Exists(linkIndex))
                    {
                        if (visited.Add(linkIndex))
                        {
                            sb.Append('(');
                            var link = new UInt64Link(links.GetLink(linkIndex));
                            if (renderIndex)
                            {
                                sb.Append(link.Index);
                                sb.Append(':');
                            }
                            if (link.Source == link.Index)
                            {
                                sb.Append(link.Index);
                            }
                            else
                            {
                                var source = new UInt64Link(links.GetLink(link.Source));
                                if (isElement(source))
                                {
                                    appendElement(sb, source);
                                }
                                else
                                {
                                    links.AppendStructure(sb, visited, source.Index, isElement,
                                    ↪  appendElement, renderIndex);
                                }
                            }
                            sb.Append(' ');
                            if (link.Target == link.Index)
                            {
                                sb.Append(link.Index);
```

```
123                         }
124                     else
125                     {
126                         var target = new UInt64Link(links.GetLink(link.Target));
127                         if (isElement(target))
128                         {
129                             appendElement(sb, target);
130                         }
131                         else
132                         {
133                             links.AppendStructure(sb, visited, target.Index, isElement,
     ↪    appendElement, renderIndex);
134                         }
135                     }
136                     sb.Append(')');
137                 }
138                 else
139                 {
140                     if (renderDebug)
141                     {
142                         sb.Append('*');
143                     }
144                     sb.Append(linkIndex);
145                 }
146             }
147             else
148             {
149                 if (renderDebug)
150                 {
151                     sb.Append('~');
152                 }
153                 sb.Append(linkIndex);
154             }
155         }
156     }
157 }
```

./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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16  namespace Platform.Data.Doublets
17  {
18      public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
19      {
20          /// <remarks>
21          /// Альтернативные варианты хранения трансформации (элемента транзакции):
22          ///
23          /// private enum TransitionType
24          /// {
25          ///     Creation,
26          ///     UpdateOf,
27          ///     UpdateTo,
28          ///     Deletion
29          /// }
30          ///
31          /// private struct Transition
32          /// {
33          ///     public ulong TransactionId;
34          ///     public UniqueTimestamp Timestamp;
35          ///     public TransactionItemType Type;
36          ///     public Link Source;
37          ///     public Link Linker;
38          ///     public Link Target;
39          /// }
40          ///
41          /// Или
```

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

```csharp
            ///      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:
            ///      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;
```

```csharp
                            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)
                    {
                        Revert();
                    }
                    _layer.ResetCurrentTransation();
                }
            }
        }

        public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);

        private readonly string _logAddress;
        private readonly FileStream _log;
        private readonly Queue<Transition> _transitions;
        private readonly UniqueTimestampFactory _uniqueTimestampFactory;
        private Task _transitionsPusher;
        private Transition _lastCommitedTransition;
        private ulong _currentTransactionId;
        private Queue<Transition> _currentTransactionTransitions;
        private Transaction _currentTransaction;
        private ulong _lastCommitedTransactionId;

        public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
            : base(links)
        {
            if (string.IsNullOrWhiteSpace(logAddress))
            {
                throw new ArgumentNullException(nameof(logAddress));
            }
            // В первой строке файла хранится последняя закоммиченную транзакцию.
            // При запуске это используется для проверки удачного закрытия файла лога.
            // In the first line of the file the last committed transaction is stored.
            // On startup, this is used to check that the log file is successfully closed.
            var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
            var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
            if (!lastCommitedTransition.Equals(lastWrittenTransition))
            {
                Dispose();
                throw new NotSupportedException("Database is damaged, autorecovery is not
                    supported yet.");
            }
            if (lastCommitedTransition.Equals(default(Transition)))
            {
                FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
            }
            _lastCommitedTransition = lastCommitedTransition;
            // TODO: Think about a better way to calculate or store this value
            var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
            _lastCommitedTransactionId = allTransitions.Max(x => x.TransactionId);
            _uniqueTimestampFactory = new UniqueTimestampFactory();
            _logAddress = logAddress;
            _log = FileHelpers.Append(logAddress);
            _transitions = new Queue<Transition>();
            _transitionsPusher = new Task(TransitionsPusher);
            _transitionsPusher.Start();
        }

        public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
```

```csharp
268    public override ulong Create()
269    {
270        var createdLinkIndex = Links.Create();
271        var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
272        CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
       ↪   default, createdLink));
273        return createdLinkIndex;
274    }
275
276    public override ulong Update(IList<ulong> parts)
277    {
278        var linkIndex = parts[Constants.IndexPart];
279        var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
280        linkIndex = Links.Update(parts);
281        var afterLink = new UInt64Link(Links.GetLink(linkIndex));
282        CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
       ↪   beforeLink, afterLink));
283        return linkIndex;
284    }
285
286    public override void Delete(ulong link)
287    {
288        var deletedLink = new UInt64Link(Links.GetLink(link));
289        Links.Delete(link);
290        CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
       ↪   deletedLink, default));
291    }
292
293    [MethodImpl(MethodImplOptions.AggressiveInlining)]
294    private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
       ↪   _transitions;
295
296    private void CommitTransition(Transition transition)
297    {
298        if (_currentTransaction != null)
299        {
300            Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
301        }
302        var transitions = GetCurrentTransitions();
303        transitions.Enqueue(transition);
304    }
305
306    private void RevertTransition(Transition transition)
307    {
308        if (transition.After.IsNull()) // Revert Deletion with Creation
309        {
310            Links.Create();
311        }
312        else if (transition.Before.IsNull()) // Revert Creation with Deletion
313        {
314            Links.Delete(transition.After.Index);
315        }
316        else // Revert Update
317        {
318            Links.Update(new[] { transition.After.Index, transition.Before.Source,
           ↪   transition.Before.Target });
319        }
320    }
321
322    private void ResetCurrentTransation()
323    {
324        _currentTransactionId = 0;
325        _currentTransactionTransitions = null;
326        _currentTransaction = null;
327    }
328
329    private void PushTransitions()
330    {
331        if (_log == null || _transitions == null)
332        {
333            return;
334        }
335        for (var i = 0; i < _transitions.Count; i++)
336        {
337            var transition = _transitions.Dequeue();
338
339            _log.Write(transition);
340            _lastCommitedTransition = transition;
```

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

./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs

```
1   using Platform.Interfaces;
2   using Platform.Numbers;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Unicode
7   {
8       public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
        ↪   IConverter<char, TLink>
9       {
10          private readonly IConverter<TLink> _addressToNumberConverter;
11          private readonly TLink _unicodeSymbolMarker;
12
13          public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
            ↪   addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
14          {
15              _addressToNumberConverter = addressToNumberConverter;
16              _unicodeSymbolMarker = unicodeSymbolMarker;
17          }
18
19          public TLink Convert(char source)
20          {
21              var unaryNumber = _addressToNumberConverter.Convert((Integer<TLink>)source);
22              return Links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
23          }
24      }
25  }
```

```
1   using Platform.Data.Doublets.Sequences.Indexes;
2   using Platform.Interfaces;
3   using System.Collections.Generic;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Unicode
8   {
9       public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<string, TLink>
10      {
11          private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
12          private readonly ISequenceIndex<TLink> _index;
13          private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
14          private readonly TLink _unicodeSequenceMarker;
15
16          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
            ↪  charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
            ↪  TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
17          {
18              _charToUnicodeSymbolConverter = charToUnicodeSymbolConverter;
19              _index = index;
20              _listToSequenceLinkConverter = listToSequenceLinkConverter;
21              _unicodeSequenceMarker = unicodeSequenceMarker;
22          }
23
24          public TLink Convert(string source)
25          {
26              var elements = new List<TLink>();
27              for (int i = 0; i < source.Length; i++)
28              {
29                  elements.Add(_charToUnicodeSymbolConverter.Convert(source[i]));
30              }
31              _index.Add(elements);
32              var sequence = _listToSequenceLinkConverter.Convert(elements);
33              return Links.GetOrCreate(sequence, _unicodeSequenceMarker);
34          }
35      }
36  }
```

```
1   using System;
2   using System.Collections.Generic;
3   using System.Globalization;
4   using System.Runtime.CompilerServices;
5   using System.Text;
6   using Platform.Data.Sequences;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets.Unicode
11  {
12      public class UnicodeMap
13      {
14          public static readonly ulong FirstCharLink = 1;
15          public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
16          public static readonly ulong MapSize = 1 + char.MaxValue;
17
18          private readonly ILinks<ulong> _links;
19          private bool _initialized;
20
21          public UnicodeMap(ILinks<ulong> links) => _links = links;
22
23          public static UnicodeMap InitNew(ILinks<ulong> links)
24          {
25              var map = new UnicodeMap(links);
26              map.Init();
27              return map;
28          }
29
30          public void Init()
31          {
32              if (_initialized)
33              {
34                  return;
35              }
36              _initialized = true;
37              var firstLink = _links.CreatePoint();
38              if (firstLink != FirstCharLink)
39              {
```

```csharp
                    _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)
        {
            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++)
```

```
114                 {
115                     linksSequence[i] = FromCharToLink(sequence[i]);
116                 }
117                 return linksSequence;
118             }
119
120         public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
121             {
122                 var result = new List<ulong[]>();
123                 var offset = 0;
124                 while (offset < sequence.Length)
125                 {
126                     var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
127                     var relativeLength = 1;
128                     var absoluteLength = offset + relativeLength;
129                     while (absoluteLength < sequence.Length &&
130                             currentCategory ==
                            ↪   CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
131                     {
132                         relativeLength++;
133                         absoluteLength++;
134                     }
135                     // char array to ulong array
136                     var innerSequence = new ulong[relativeLength];
137                     var maxLength = offset + relativeLength;
138                     for (var i = offset; i < maxLength; i++)
139                     {
140                         innerSequence[i - offset] = FromCharToLink(sequence[i]);
141                     }
142                     result.Add(innerSequence);
143                     offset += relativeLength;
144                 }
145                 return result;
146             }
147
148         public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
149             {
150                 var result = new List<ulong[]>();
151                 var offset = 0;
152                 while (offset < array.Length)
153                 {
154                     var relativeLength = 1;
155                     if (array[offset] <= LastCharLink)
156                     {
157                         var currentCategory =
                            ↪   CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
158                         var absoluteLength = offset + relativeLength;
159                         while (absoluteLength < array.Length &&
160                                 array[absoluteLength] <= LastCharLink &&
161                                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(↵
                                ↪   array[absoluteLength])))
162                         {
163                             relativeLength++;
164                             absoluteLength++;
165                         }
166                     }
167                     else
168                     {
169                         var absoluteLength = offset + relativeLength;
170                         while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
171                         {
172                             relativeLength++;
173                             absoluteLength++;
174                         }
175                     }
176                     // copy array
177                     var innerSequence = new ulong[relativeLength];
178                     var maxLength = offset + relativeLength;
179                     for (var i = offset; i < maxLength; i++)
180                     {
181                         innerSequence[i - offset] = array[i];
182                     }
183                     result.Add(innerSequence);
184                     offset += relativeLength;
185                 }
186                 return result;
187             }
188         }
189     }
```

./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs

```csharp
1   using Platform.Interfaces;
2   using System.Collections.Generic;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Unicode
7   {
8       public class UnicodeSequenceCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
        ↪ ICriterionMatcher<TLink>
9       {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
11          private readonly TLink _unicodeSequenceMarker;
12          public UnicodeSequenceCriterionMatcher(ILinks<TLink> links, TLink unicodeSequenceMarker)
            ↪ : base(links) => _unicodeSequenceMarker = unicodeSequenceMarker;
13          public bool IsMatched(TLink link) => _equalityComparer.Equals(Links.GetTarget(link),
            ↪ _unicodeSequenceMarker);
14      }
15  }
```

./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs

```csharp
1   using System;
2   using System.Linq;
3   using Platform.Data.Doublets.Sequences.Walkers;
4   using Platform.Interfaces;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Unicode
9   {
10      public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
        ↪ IConverter<TLink, string>
11      {
12          private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
13          private readonly ISequenceWalker<TLink> _sequenceWalker;
14          private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;
15
16          public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
            ↪ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
            ↪ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
17          {
18              _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
19              _sequenceWalker = sequenceWalker;
20              _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
21          }
22
23          public string Convert(TLink source)
24          {
25              if(!_unicodeSequenceCriterionMatcher.IsMatched(source))
26              {
27                  throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                    ↪ not a unicode sequence.");
28              }
29              var sequence = Links.GetSource(source);
30              var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
                ↪ Convert).ToArray();
31              return new string(charArray);
32          }
33      }
34  }
```

./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs

```csharp
1   using Platform.Interfaces;
2   using System.Collections.Generic;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Unicode
7   {
8       public class UnicodeSymbolCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
        ↪ ICriterionMatcher<TLink>
9       {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
11          private readonly TLink _unicodeSymbolMarker;
12          public UnicodeSymbolCriterionMatcher(ILinks<TLink> links, TLink unicodeSymbolMarker) :
            ↪ base(links) => _unicodeSymbolMarker = unicodeSymbolMarker;
13          public bool IsMatched(TLink link) => _equalityComparer.Equals(Links.GetTarget(link),
            ↪ _unicodeSymbolMarker);
```

```
14        }
15    }
```

./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs

```csharp
1    using System;
2    using Platform.Interfaces;
3    using Platform.Numbers;
4
5    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7    namespace Platform.Data.Doublets.Unicode
8    {
9        public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
         ↪   IConverter<TLink, char>
10       {
11           private readonly IConverter<TLink> _numberToAddressConverter;
12           private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
13
14           public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
            ↪   numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
            ↪   base(links)
15           {
16               _numberToAddressConverter = numberToAddressConverter;
17               _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
18           }
19
20           public char Convert(TLink source)
21           {
22               if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
23               {
24                   throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                    ↪   not a unicode symbol.");
25               }
26               return (char)(ushort)(Integer<TLink>)_numberToAddressConverter.Convert(Links.GetSour
                 ↪   ce(source));
27           }
28       }
29   }
```

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

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

54
55            var comparer2 = new UInt64Comparer();

56
57            var ts4 = Performance.Measure(() =>
58            {
59                for (int i = 0; i < N; i++)
60                {
61                    result = comparer2.Compare(x, y) >= 0;
62                }
63            });

64
65            Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");
66        }
67    }
68 }
```

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

```
1  using System.Collections.Generic;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Numbers;
5  using Platform.Memory;
6  using Platform.Scopes;
7  using Platform.Setters;
8  using Platform.Data.Doublets.ResizableDirectMemory;

9
10 namespace Platform.Data.Doublets.Tests
11 {
12     public static class DoubletLinksTests
13     {
14         [Fact]
15         public static void UInt64CRUDTest()
16         {
17             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                 ↪  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             {
```

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

```csharp
                    scope.Use<ILinks<uint>>().TestRawNumbersCRUDOperations();
            }
        }

        [Fact]
        public static void UInt16RawNumbersCRUDTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪ ResizableDirectMemoryLinks<ushort>>>())
            {
                scope.Use<ILinks<ushort>>().TestRawNumbersCRUDOperations();
            }
        }

        [Fact]
        public static void UInt8RawNumbersCRUDTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪ ResizableDirectMemoryLinks<byte>>>())
            {
                scope.Use<ILinks<byte>>().TestRawNumbersCRUDOperations();
            }
        }

        private static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
        {
            // Constants
            var constants = links.Constants;
            var equalityComparer = EqualityComparer<T>.Default;

            var h106E = new Hybrid<T>(106L, isExternal: true);
            var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
            var h108E = new Hybrid<T>(-108L);

            Assert.Equal(106L, h106E.AbsoluteValue);
            Assert.Equal(107L, h107E.AbsoluteValue);
            Assert.Equal(108L, h108E.AbsoluteValue);

            // Create Link (External -> External)
            var linkAddress1 = links.Create();

            links.Update(linkAddress1, h106E, h108E);

            var link1 = new Link<T>(links.GetLink(linkAddress1));

            Assert.True(equalityComparer.Equals(link1.Source, h106E));
            Assert.True(equalityComparer.Equals(link1.Target, h108E));

            // Create Link (Internal -> External)
            var linkAddress2 = links.Create();

            links.Update(linkAddress2, linkAddress1, h108E);

            var link2 = new Link<T>(links.GetLink(linkAddress2));

            Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
            Assert.True(equalityComparer.Equals(link2.Target, h108E));

            // Create Link (Internal -> Internal)
            var linkAddress3 = links.Create();

            links.Update(linkAddress3, linkAddress1, linkAddress2);

            var link3 = new Link<T>(links.GetLink(linkAddress3));

            Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
            Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));

            // Search for created link
            var setter1 = new Setter<T>(constants.Null);
            links.Each(h106E, h108E, setter1.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));

            // Search for nonexistent link
            var setter2 = new Setter<T>(constants.Null);
            links.Each(h106E, h107E, setter2.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
```

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

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

```
1    using System;
2    using System.Collections.Generic;
3    using Xunit;
4    using Platform.Diagnostics;
5
6    namespace Platform.Data.Doublets.Tests
7    {
8        public static class EqualityTests
9        {
10           protected class UInt64EqualityComparer : IEqualityComparer<ulong>
11           {
12               public bool Equals(ulong x, ulong y) => x == y;
13
14               public int GetHashCode(ulong obj) => obj.GetHashCode();
15           }
16
17           private static bool Equals1<T>(T x, T y) => Equals(x, y);
18
19           private static bool Equals2<T>(T x, T y) => x.Equals(y);
20
21           private static bool Equals3(ulong x, ulong y) => x == y;
22
23           [Fact]
24           public static void 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             });

57             var equalityComparer1 = EqualityComparer<ulong>.Default;

59             var ts4 = Performance.Measure(() =>
60             {
61                 for (int i = 0; i < N; i++)
62                 {
63                     result = equalityComparer1.Equals(x, y);
64                 }
65             });

67             var equalityComparer2 = new UInt64EqualityComparer();

69             var ts5 = Performance.Measure(() =>
70             {
71                 for (int i = 0; i < N; i++)
72                 {
73                     result = equalityComparer2.Equals(x, y);
74                 }
75             });

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

79             var ts6 = Performance.Measure(() =>
80             {
81                 for (int i = 0; i < N; i++)
82                 {
83                     result = equalityComparer3(x, y);
84                 }
85             });

87             var comparer = Comparer<ulong>.Default;

89             var ts7 = Performance.Measure(() =>
90             {
91                 for (int i = 0; i < N; i++)
92                 {
93                     result = comparer.Compare(x, y) == 0;
94                 }
95             });

97             Assert.True(ts2 < ts1);
98             Assert.True(ts3 < ts2);
99             Assert.True(ts5 < ts4);
100            Assert.True(ts5 < ts6);

102            Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");
103        }
104    }
105 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.IO;
5  using System.Text;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Xunit;
9  using Platform.Disposables;
10 using Platform.IO;
11 using Platform.Ranges;
12 using Platform.Random;
13 using Platform.Timestamps;
14 using Platform.Singletons;
15 using Platform.Counters;
16 using Platform.Diagnostics;
17 using Platform.Data.Constants;
18 using Platform.Data.Doublets.ResizableDirectMemory;
19 using Platform.Data.Doublets.Decorators;

21 namespace Platform.Data.Doublets.Tests
22 {
23     public static class LinksTests
24     {
25         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
             ↪  Default<LinksCombinedConstants<bool, ulong, int>>.Instance;

26
```

```csharp
        private const long Iterations = 10 * 1024;

        #region Concept

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

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

                for (var N = 0; N < 100; N++)
                {
                    var random = new System.Random(N);

                    var created = 0;
                    var deleted = 0;

                    for (var i = 0; i < N; i++)
                    {
                        var linksCount = links.Count();

                        var createPoint = random.NextBoolean();

                        if (linksCount > 2 && createPoint)
                        {
                            var linksAddressRange = new Range<ulong>(1, linksCount);
                            var source = random.NextUInt64(linksAddressRange);
                            var target = random.NextUInt64(linksAddressRange); //-V3086

                            var resultLink = links.CreateAndUpdate(source, target);
                            if (resultLink > linksCount)
                            {
                                created++;
                            }
                        }
                        else
                        {
                            links.Create();
                            created++;
                        }
                    }

                    Assert.True(created == (int)links.Count());

                    for (var i = 0; i < N; i++)
                    {
                        var link = (ulong)i + 1;
                        if (links.Exists(link))
                        {
                            links.Delete(link);
                            deleted++;
                        }
                    }

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

        [Fact]
        public static void CascadeUpdateTest()
        {
            var itself = _constants.Itself;

            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;

                var l1 = links.Create();
                var l2 = links.Create();

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

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

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

```csharp
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                }
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);
```

```csharp
                            links.CreateAndUpdate(l2, itself);
                            links.CreateAndUpdate(l2, itself);

                            //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi⤸
                            ↪   tion>(scope.TempTransactionLogFilename);

                            l2 = links.Update(l2, l1);

                            links.Delete(l2);

                            ExceptionThrower();

                            transaction.Commit();
                        }

                        Global.Trash = links.Count();
                    }
                }
                catch
                {
                    Assert.False(lastScope == null);

                    var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l⤸
                    ↪   astScope.TempTransactionLogFilename);

                    Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
                    ↪   transitions[0].After.IsNull());

                    lastScope.DeleteFiles();
                }
            }

            [Fact]
            public static void TransactionUserCodeErrorSomeDataSavedTest()
            {
                // User Code Error (Autoreverted), some data saved
                var itself = _constants.Itself;

                TempLinksTestScope lastScope = null;
                try
                {
                    ulong l1;
                    ulong l2;

                    using (var scope = new TempLinksTestScope(useLog: true))
                    {
                        var links = scope.Links;
                        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);

                        links.Unsync.DisposeIfPossible();

                        Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(⤸
                        ↪   scope.TempTransactionLogFilename);
                    }

                    using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪   useLog: true))
                    {
                        var links = scope.Links;
                        var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
                        using (var transaction = transactionsLayer.BeginTransaction())
                        {
                            l2 = links.Update(l2, l1);

                            links.Delete(l2);

                            ExceptionThrower();

                            transaction.Commit();
                        }

                        Global.Trash = links.Count();
                    }
```

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

```csharp
                FileHelpers.WriteFirst(tempTransactionLogFilename, new
                ↪   UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));

            // Try load damaged database
            try
            {
                // TODO: Fix
                using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪   UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                ↪   tempTransactionLogFilename))
                using (var links = new UInt64Links(memoryAdapter))
                {
                    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
```

```csharp
        {
            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp⌋
            ↪   TransactionLogFilename);
        }

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

    private static void ExceptionThrower()
    {
        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());
    }
```

```csharp
        #endregion

        #region Performance

         /*
        public static void RunAllPerformanceTests()
        {
            try
            {
                links.TestLinksInSteps();
            }
            catch (Exception ex)
            {
                ex.WriteToConsole();
            }

            return;

            try
            {
                //ThreadPool.SetMaxThreads(2, 2);

                // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    результат
                // Также это дополнительно помогает в отладке
                // Увеличивает вероятность попадания информации в кэши
                for (var i = 0; i < 10; i++)
                {
                    //0 - 10 ГБ
                    //Каждые 100 МБ срез цифр

                    //links.TestGetSourceFunction();
                    //links.TestGetSourceFunctionInParallel();
                    //links.TestGetTargetFunction();
                    //links.TestGetTargetFunctionInParallel();
                    links.Create64BillionLinks();

                    links.TestRandomSearchFixed();
                    //links.Create64BillionLinksInParallel();
                    links.TestEachFunction();
                    //links.TestForeach();
                    //links.TestParallelForeach();
                }

                links.TestDeletionOfAllLinks();

            }
            catch (Exception ex)
            {
                ex.WriteToConsole();
            }
        }*/

         /*
        public static void TestLinksInSteps()
        {
            const long gibibyte = 1024 * 1024 * 1024;
            const long mebibyte = 1024 * 1024;

            var totalLinksToCreate = gibibyte /
    Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
            var linksStep = 102 * mebibyte /
    Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;

            var creationMeasurements = new List<TimeSpan>();
            var searchMeasuremets = new List<TimeSpan>();
            var deletionMeasurements = new List<TimeSpan>();

            GetBaseRandomLoopOverhead(linksStep);
            GetBaseRandomLoopOverhead(linksStep);

            var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);

            ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);

            var loops = totalLinksToCreate / linksStep;

            for (int i = 0; i < loops; i++)
            {
```

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

```csharp
                    // 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;
```

```
694
695                    links.Delete(firstLink);
696
697                    ConsoleHelpers.Debug(
698                        "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪ second), counter result: {3}",
699                        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
700                }
701            }
702
703        [Fact(Skip = "performance test")]
704        public static void TestGetTargetInParallel()
705        {
706            using (var scope = new TempLinksTestScope())
707            {
708                var links = scope.Links;
709                ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
                ↪ 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      {
775          using (var scope = new TempLinksTestScope())
776          {
777              var links = scope.Links;
778              ulong counter = 0;
779
780              var maxLink = links.Count();
781
782              var iterations = links.Count();
783
784              ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
                  ↪  links.Count());
785
786              var sw = Stopwatch.StartNew();
787
788              for (var i = iterations; i > 0; i--)
789              {
790                  var linksAddressRange = new Range<ulong>(_constants.MinPossibleIndex,
                      ↪  maxLink);
791
792                  var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
793                  var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
794
795                  counter += links.SearchOrDefault(source, target);
796              }
797
798              var elapsedTime = sw.Elapsed;
799
800              var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
801
802              ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
                  ↪  Iterations per second), c: {3}",
803                      iterations, elapsedTime, (long)iterationsPerSecond, counter);
804          }
805      }
806
807      [Fact(Skip = "useless: O(0), was dependent on creation tests")]
808      public static void TestEach()
809      {
810          using (var scope = new TempLinksTestScope())
811          {
812              var links = scope.Links;
813
814              var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
815
816              ConsoleHelpers.Debug("Testing Each function.");
817
818              var sw = Stopwatch.StartNew();
819
820              links.Each(counter.IncrementAndReturnTrue);
821
822              var elapsedTime = sw.Elapsed;
823
824              var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
825
826              ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
                  ↪  links per second)",
827                      counter, elapsedTime, (long)linksPerSecond);
828          }
829      }
830
831      /*
832      [Fact]
833      public static void TestForeach()
834      {
835          var tempFilename = Path.GetTempFileName();
836
837          using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
```

```csharp
            {
                ulong counter = 0;

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

                var sw = Stopwatch.StartNew();

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

                var elapsedTime = sw.Elapsed;

                var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;

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

            File.Delete(tempFilename);
        }
        */

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

            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename, DefaultLinksSizeStep))
            {

                long counter = 0;

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

                var sw = Stopwatch.StartNew();

                //Parallel.ForEach((IEnumerable<ulong>)links, x =>
                //{
                //    Interlocked.Increment(ref counter);
                //});

                var elapsedTime = sw.Elapsed;

                var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;

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

            File.Delete(tempFilename);
        }
        */

        [Fact(Skip = "performance test")]
        public static void Create64BillionLinks()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var linksBeforeTest = links.Count();

                long linksToCreate = 64 * 1024 * 1024 /
                    UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;

                ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);

                var elapsedTime = Performance.Measure(() =>
                {
                    for (long i = 0; i < linksToCreate; i++)
                    {
                        links.Create();
                    }
                });

                var linksCreated = links.Count() - linksBeforeTest;
                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.Data.Doublets.Sequences;
6  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8  using Platform.Data.Doublets.Sequences.Converters;
9  using Platform.Data.Doublets.PropertyOperators;
10 using Platform.Data.Doublets.Incrementers;
11 using Platform.Data.Doublets.Sequences.Walkers;
12 using Platform.Data.Doublets.Sequences.Indexes;
13 using Platform.Data.Doublets.Unicode;
14 using Platform.Data.Doublets.Numbers.Unary;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class OptimalVariantSequenceTests
```

```csharp
    {
        private const string SequenceExample = "зеленела зелёная зелень";

        [Fact]
        public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;
                var constants = links.Constants;

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);

                var meaningRoot = links.CreatePoint();
                var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
                var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
                var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                →   constants.Itself);

                var unaryNumberToAddressConverter = new
                →   UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
                var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
                var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                →   frequencyMarker, unaryOne, unaryNumberIncrementer);
                var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                →   frequencyPropertyMarker, frequencyMarker);
                var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                →   frequencyPropertyOperator, frequencyIncrementer);
                var linkToItsFrequencyNumberConverter = new
                →   LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                →   unaryNumberToAddressConverter);
                var sequenceToItsLocalElementLevelsConverter = new
                →   SequenceToItsLocalElementLevelsConverter<ulong>(links,
                →   linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                →   sequenceToItsLocalElementLevelsConverter);

                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                →   Walker = new LeveledSequenceWalker<ulong>(links) });

                ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                →   index, optimalVariantConverter);
            }
        }

        [Fact]
        public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);

                var linksToFrequencies = new Dictionary<ulong, ulong>();

                var totalSequenceSymbolFrequencyCounter = new
                →   TotalSequenceSymbolFrequencyCounter<ulong>(links);

                var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                →   totalSequenceSymbolFrequencyCounter);

                var index = new
                →   CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
                var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque⌋
                →   ncyNumberConverter<ulong>(linkFrequenciesCache);

                var sequenceToItsLocalElementLevelsConverter = new
                →   SequenceToItsLocalElementLevelsConverter<ulong>(links,
                →   linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                →   sequenceToItsLocalElementLevelsConverter);
```

```
77          var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
            ↪  Walker = new LeveledSequenceWalker<ulong>(links) });
78
79          ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
            ↪  index, optimalVariantConverter);
80      }
81  }
82
83  private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
    ↪  SequenceToItsLocalElementLevelsConverter<ulong>
    ↪  sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
    ↪  OptimalVariantConverter<ulong> optimalVariantConverter)
84  {
85      index.Add(sequence);
86
87      var optimalVariant = optimalVariantConverter.Convert(sequence);
88
89      var readSequence1 = sequences.ToList(optimalVariant);
90
91      Assert.True(sequence.SequenceEqual(readSequence1));
92  }
93  }
94 }
```

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

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

```
54          Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
            ↪  {sw2.Elapsed}");
55
56          for (var i = 0; i < sequenceLength; i++)
57          {
58              links.Delete(sequence[i]);
59          }
60      }
61   }
62  }
63 }
```

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

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

```
62              Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
63              memoryAdapter.Delete(link);
64          }
65      }
66  }
```

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

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

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

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

```csharp
27          }
28
29          [Fact]
30          public static void CreateAllVariantsTest()
31          {
32              const long sequenceLength = 8;
33
34              using (var scope = new TempLinksTestScope(useSequences: true))
35              {
36                  var links = scope.Links;
37                  var sequences = scope.Sequences;
38
39                  var sequence = new ulong[sequenceLength];
40                  for (var i = 0; i < sequenceLength; i++)
41                  {
42                      sequence[i] = links.Create();
43                  }
44
45                  var sw1 = Stopwatch.StartNew();
46                  var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
47
48                  var sw2 = Stopwatch.StartNew();
49                  var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
50
51                  Assert.True(results1.Count > results2.Length);
52                  Assert.True(sw1.Elapsed > sw2.Elapsed);
53
54                  for (var i = 0; i < sequenceLength; i++)
55                  {
56                      links.Delete(sequence[i]);
57                  }
58
59                  Assert.True(links.Count() == 0);
60              }
61          }
62
63          //[Fact]
64          //public void CUDTest()
65          //{
66          //    var tempFilename = Path.GetTempFileName();
67
68          //    const long sequenceLength = 8;
69
70          //    const ulong itself = LinksConstants.Itself;
71
72          //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
             ↪  DefaultLinksSizeStep))
73          //    using (var links = new Links(memoryAdapter))
74          //    {
75          //        var sequence = new ulong[sequenceLength];
76          //        for (var i = 0; i < sequenceLength; i++)
77          //            sequence[i] = links.Create(itself, itself);
78
79
80          //        SequencesOptions o = new SequencesOptions();
81
82          // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
83          //        o.
84
85
86          //        var sequences = new Sequences(links);
87
88          //        var sw1 = Stopwatch.StartNew();
89          //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
90
91          //        var sw2 = Stopwatch.StartNew();
92          //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
93
94          //        Assert.True(results1.Count > results2.Length);
95          //        Assert.True(sw1.Elapsed > sw2.Elapsed);
96
97          //        for (var i = 0; i < sequenceLength; i++)
98          //            links.Delete(sequence[i]);
99          //    }
100
101         //    File.Delete(tempFilename);
102         //}
103
104         [Fact]
105         public static void AllVariantsSearchTest()
```

```csharp
        {
            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++)
                {
                    sequence[i] = links.Create();
                }

                var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();

                //for (int i = 0; i < createResults.Length; i++)
                //    sequences.Create(createResults[i]);

                var sw0 = Stopwatch.StartNew();
                var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();

                var sw1 = Stopwatch.StartNew();
                var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();

                var sw2 = Stopwatch.StartNew();
                var searchResults2 = sequences.Each1(sequence); sw2.Stop();

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

                var intersection0 = createResults.Intersect(searchResults0).ToList();
                Assert.True(intersection0.Count == searchResults0.Count);
                Assert.True(intersection0.Count == createResults.Length);

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

                var intersection2 = createResults.Intersect(searchResults2).ToList();
                Assert.True(intersection2.Count == searchResults2.Count);
                Assert.True(intersection2.Count == createResults.Length);

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

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

        [Fact]
        public static void BalancedVariantSearchTest()
        {
            const long sequenceLength = 200;

            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 balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

                var sw1 = Stopwatch.StartNew();
                var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();

                var sw2 = Stopwatch.StartNew();
                var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();

                var sw3 = Stopwatch.StartNew();
                var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
```

```csharp
                    // На количестве в 200 элементов это будет занимать вечность
                    //var sw4 = Stopwatch.StartNew();
                    //var searchResults4 = sequences.Each(sequence); sw4.Stop();

                    Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);

                    Assert.True(searchResults3.Count == 1 && balancedVariant ==
                    ↪  searchResults3.First());

                    //Assert.True(sw1.Elapsed < sw2.Elapsed);

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

            [Fact]
            public static void AllPartialVariantsSearchTest()
            {
                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++)
                    {
                        sequence[i] = links.Create();
                    }

                    var createResults = sequences.CreateAllVariants2(sequence);

                    //var createResultsStrings = createResults.Select(x => x + ": " +
                    ↪  sequences.FormatSequence(x)).ToList();
                    //Global.Trash = createResultsStrings;

                    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();

                    //var sw3 = Stopwatch.StartNew();
                    //var searchResults3 =
                    ↪  sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();

                    var sw4 = Stopwatch.StartNew();
                    var searchResults4 =
                    ↪  sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();

                    //Global.Trash = searchResults3;

                    //var searchResults1Strings = searchResults1.Select(x => x + ": " +
                    ↪  sequences.FormatSequence(x)).ToList();
                    //Global.Trash = searchResults1Strings;

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

                    var intersection2 = createResults.Intersect(searchResults2).ToList();
                    Assert.True(intersection2.Count == createResults.Length);

                    var intersection4 = createResults.Intersect(searchResults4).ToList();
                    Assert.True(intersection4.Count == createResults.Length);

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

```
258                }
259            }
260        }
261
262        [Fact]
263        public static void BalancedPartialVariantsSearchTest()
264        {
265            const long sequenceLength = 200;
266
267            using (var scope = new TempLinksTestScope(useSequences: true))
268            {
269                var links = scope.Links;
270                var sequences = scope.Sequences;
271
272                var sequence = new ulong[sequenceLength];
273                for (var i = 0; i < sequenceLength; i++)
274                {
275                    sequence[i] = links.Create();
276                }
277
278                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
279
280                var balancedVariant = balancedVariantConverter.Convert(sequence);
281
282                var partialSequence = new ulong[sequenceLength - 2];
283
284                Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
285
286                var sw1 = Stopwatch.StartNew();
287                var searchResults1 =
     ↪    sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
288
289                var sw2 = Stopwatch.StartNew();
290                var searchResults2 =
     ↪    sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
291
292                Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
293
294                Assert.True(searchResults2.Count == 1 && balancedVariant ==
     ↪    searchResults2.First());
295
296                for (var i = 0; i < sequenceLength; i++)
297                {
298                    links.Delete(sequence[i]);
299                }
300            }
301        }
302
303        [Fact(Skip = "Correct implementation is pending")]
304        public static void PatternMatchTest()
305        {
306            var zeroOrMany = Sequences.Sequences.ZeroOrMany;
307
308            using (var scope = new TempLinksTestScope(useSequences: true))
309            {
310                var links = scope.Links;
311                var sequences = scope.Sequences;
312
313                var e1 = links.Create();
314                var e2 = links.Create();
315
316                var sequence = new[]
317                {
318                    e1, e2, e1, e2 // mama / papa
319                };
320
321                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
322
323                var balancedVariant = balancedVariantConverter.Convert(sequence);
324
325                // 1: [1]
326                // 2: [2]
327                // 3: [1,2]
328                // 4: [1,2,1,2]
329
330                var doublet = links.GetSource(balancedVariant);
331
332                var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
333
334                Assert.True(matchedSequences1.Count == 0);
```

```csharp
335
336                var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
337
338                Assert.True(matchedSequences2.Count == 0);
339
340                var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
341
342                Assert.True(matchedSequences3.Count == 0);
343
344                var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
345
346                Assert.Contains(doublet, matchedSequences4);
347                Assert.Contains(balancedVariant, matchedSequences4);
348
349                for (var i = 0; i < sequence.Length; i++)
350                {
351                    links.Delete(sequence[i]);
352                }
353            }
354        }
355
356        [Fact]
357        public static void IndexTest()
358        {
359            using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
                    true }, useSequences: true))
360            {
361                var links = scope.Links;
362                var sequences = scope.Sequences;
363                var index = sequences.Options.Index;
364
365                var e1 = links.Create();
366                var e2 = links.Create();
367
368                var sequence = new[]
369                {
370                    e1, e2, e1, e2 // mama / papa
371                };
372
373                Assert.False(index.MightContain(sequence));
374
375                index.Add(sequence);
376
377                Assert.True(index.MightContain(sequence));
378            }
379        }
380
381        /// <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>
382        private static readonly string _exampleText =
383            @"([english
                version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
384
385 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?
386
387 [![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)
388
389 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?
390
391 [![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)
392
393 А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
        так? Инверсия? Отражение? Сумма?
394
395 [![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)
```

397 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?

398
399 [![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)

400
401 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?

402
403 [![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)

404
405 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?

406
407 [![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)

408
409 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?

410
411 [![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)

412
413 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?

414
415 [![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)

416
417 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?

418
419 [![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)

420
421 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?

422
423 [![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.c
→ om/Konard/LinksPlatform/master/doc/Intro/10.png)

424
425 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
→ рекурсии или фрактала?

426

```
[![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.githubuserc⌋
 ↪  ontent.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?

[![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)

...

[![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)";


        private static readonly string _exampleLoremIpsumText =
            @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
                ↪  incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
 ↪  consequat.";

        [Fact]
        public static void CompressionTest()
        {
            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 / template [(m/p), a] { [1] [2] [1] [2] }
                };

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
                var totalSequenceSymbolFrequencyCounter = new
                 ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
                var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
                 ↪  totalSequenceSymbolFrequencyCounter);
                var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
                 ↪  balancedVariantConverter, doubletFrequenciesCache);

                var compressedVariant = compressingConverter.Convert(sequence);

                // 1: [1]       (1->1) point
                // 2: [2]       (2->2) point
                // 3: [1,2]     (1->2) doublet
                // 4: [1,2,1,2] (3->3) doublet

                Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
                Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
                Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
                Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);

                var source = _constants.SourcePart;
                var target = _constants.TargetPart;

                Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
                Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
                Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
                Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);

                // 4 - length of sequence
                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
                 ↪  == sequence[0]);
```

```csharp
485             Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
                  ↪  == sequence[1]);
486             Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
                  ↪  == sequence[2]);
487             Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
                  ↪  == sequence[3]);
488         }
489     }
490
491     [Fact]
492     public static void CompressionEfficiencyTest()
493     {
494         var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
              ↪  StringSplitOptions.RemoveEmptyEntries);
495         var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
496         var totalCharacters = arrays.Select(x => x.Length).Sum();
497
498         using (var scope1 = new TempLinksTestScope(useSequences: true))
499         using (var scope2 = new TempLinksTestScope(useSequences: true))
500         using (var scope3 = new TempLinksTestScope(useSequences: true))
501         {
502             scope1.Links.Unsync.UseUnicode();
503             scope2.Links.Unsync.UseUnicode();
504             scope3.Links.Unsync.UseUnicode();
505
506             var balancedVariantConverter1 = new
                  ↪  BalancedVariantConverter<ulong>(scope1.Links.Unsync);
507             var totalSequenceSymbolFrequencyCounter = new
                  ↪  TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
508             var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
                  ↪  totalSequenceSymbolFrequencyCounter);
509             var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
                  ↪  balancedVariantConverter1, linkFrequenciesCache1,
                  ↪  doInitialFrequenciesIncrement: false);
510
511             var compressor2 = scope2.Sequences;
512             var compressor3 = scope3.Sequences;
513
514             var constants = Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
515
516             var sequences = compressor3;
517             //var meaningRoot = links.CreatePoint();
518             //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
519             //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
520             //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                  ↪  constants.Itself);
521
522             //var unaryNumberToAddressConverter = new
                  ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
523             //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
                  ↪  unaryOne);
524             //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                  ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
525             //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
                  ↪  frequencyPropertyMarker, frequencyMarker);
526             //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                  ↪  frequencyPropertyOperator, frequencyIncrementer);
527             //var linkToItsFrequencyNumberConverter = new
                  ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                  ↪  unaryNumberToAddressConverter);
528
529             var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
                  ↪  totalSequenceSymbolFrequencyCounter);
530
531             var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque⌋
                  ↪  ncyNumberConverter<ulong>(linkFrequenciesCache3);
532
533             var sequenceToItsLocalElementLevelsConverter = new
                  ↪  SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
                  ↪  linkToItsFrequencyNumberConverter);
534             var optimalVariantConverter = new
                  ↪  OptimalVariantConverter<ulong>(scope3.Links.Unsync,
                  ↪  sequenceToItsLocalElementLevelsConverter);
535
536             var compressed1 = new ulong[arrays.Length];
537             var compressed2 = new ulong[arrays.Length];
538             var compressed3 = new ulong[arrays.Length];
```

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

            //for (int i = START; i < END; i++)
            //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);

            var initialCount1 = scope2.Links.Unsync.Count();

            var sw1 = Stopwatch.StartNew();

            for (int i = START; i < END; i++)
            {
                linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
                compressed1[i] = compressor1.Convert(arrays[i]);
            }

            var elapsed1 = sw1.Elapsed;

            var balancedVariantConverter2 = new
                BalancedVariantConverter<ulong>(scope2.Links.Unsync);

            var initialCount2 = scope2.Links.Unsync.Count();

            var sw2 = Stopwatch.StartNew();

            for (int i = START; i < END; i++)
            {
                compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
            }

            var elapsed2 = sw2.Elapsed;

            for (int i = START; i < END; i++)
            {
                linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
            }

            var initialCount3 = scope3.Links.Unsync.Count();

            var sw3 = Stopwatch.StartNew();

            for (int i = START; i < END; i++)
            {
                //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
                compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
            }

            var elapsed3 = sw3.Elapsed;

            Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
                Optimal variant: {elapsed3}");

            // Assert.True(elapsed1 > elapsed2);

            // Checks
            for (int i = START; i < END; i++)
            {
                var sequence1 = compressed1[i];
                var sequence2 = compressed2[i];
                var sequence3 = compressed3[i];

                var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                    scope1.Links.Unsync);

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

                var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
                    scope3.Links.Unsync);

                var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
                    link.IsPartialPoint());
                var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
                    link.IsPartialPoint());
                var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
                    link.IsPartialPoint());
```

```csharp
                            //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                            ↪  arrays[i].Length > 3)
                            //    Assert.False(structure1 == structure2);
                            //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
                            ↪  arrays[i].Length > 3)
                            //    Assert.False(structure3 == structure2);

                            Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
                            Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
                        }

                        Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
                        ↪  totalCharacters);
                        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();
```

```csharp
           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];
```

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

                        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}");

                Assert.True(scope1.Links.Count() <= scope2.Links.Count());

                //compressor1.ValidateFrequencies();
            }
        }

        [Fact]
        public static void RandomNumbersCompressionQualityTest()
        {
            const ulong N = 500;

            //const ulong minNumbers = 10000;
            //const ulong maxNumbers = 20000;

            //var strings = new List<string>();

            //for (ulong i = 0; i < N; i++)
            //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
                maxNumbers).ToString());

            var strings = new List<string>();

            for (ulong i = 0; i < N; i++)
            {
                strings.Add(RandomHelpers.Default.NextUInt64().ToString());
            }

            strings = strings.Distinct().ToList();

            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 = 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;
```

```csharp
                    for (int i = START; i < END; i++)
                    {
                        compressed1[i] = compressor1.Create(arrays[i]);
                    }

                    var elapsed1 = sw1.Elapsed;

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

                    var sw2 = Stopwatch.StartNew();

                    for (int i = START; i < END; i++)
                    {
                        compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
                    }

                    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;
```

```csharp
                        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;

                        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);

                    var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
                    Assert.True(intersection1.Count == linksTotalUsages2.Length);
                }

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

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

```
1  using System.IO;
2  using Platform.Disposables;
3  using Platform.Data.Doublets.ResizableDirectMemory;
4  using Platform.Data.Doublets.Sequences;
5  using Platform.Data.Doublets.Decorators;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public class TempLinksTestScope : DisposableBase
10     {
11         public readonly ILinks<ulong> MemoryAdapter;
12         public readonly SynchronizedLinks<ulong> Links;
13         public readonly Sequences.Sequences Sequences;
14         public readonly string TempFilename;
15         public readonly string TempTransactionLogFilename;
16         private readonly bool _deleteFiles;
17
18         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
           ↪  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 }
```

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

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

./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs

```
1  using Xunit;
2  using Platform.Interfaces;
3  using Platform.Memory;
4  using Platform.Reflection;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.Incrementers;
7  using Platform.Data.Doublets.Numbers.Raw;
8  using Platform.Data.Doublets.Numbers.Unary;
9  using Platform.Data.Doublets.PropertyOperators;
10 using Platform.Data.Doublets.ResizableDirectMemory;
11 using Platform.Data.Doublets.Sequences.Converters;
12 using Platform.Data.Doublets.Sequences.Indexes;
13 using Platform.Data.Doublets.Sequences.Walkers;
14 using Platform.Data.Doublets.Unicode;
15
16 namespace Platform.Data.Doublets.Tests
```

```csharp
{
    public static class UnicodeConvertersTests
    {
        [Fact]
        public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var meaningRoot = links.CreatePoint();
                var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
                var powerOf2ToUnaryNumberConverter = new
                    PowerOf2ToUnaryNumberConverter<ulong>(links, one);
                var addressToUnaryNumberConverter = new
                    AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var unaryNumberToAddressConverter = new
                    UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    powerOf2ToUnaryNumberConverter);
                TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                    addressToUnaryNumberConverter, unaryNumberToAddressConverter);
            }
        }

        [Fact]
        public static void CharAndRawNumberUnicodeSymbolConvertersTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ResizableDirectMemoryLinks<ulong>>>())
            {
                var links = scope.Use<ILinks<ulong>>();
                var meaningRoot = links.CreatePoint();
                var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
                var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
                TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                    addressToRawNumberConverter, rawNumberToAddressConverter);
            }
        }

        private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
            meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
            numberToAddressConverter)
        {
            var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
            var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
                addressToNumberConverter, unicodeSymbolMarker);
            var originalCharacter = 'H';
            var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
            var unicodeSymbolCriterionMatcher = new UnicodeSymbolCriterionMatcher<ulong>(links,
                unicodeSymbolMarker);
            var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
                numberToAddressConverter, unicodeSymbolCriterionMatcher);
            var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
            Assert.Equal(originalCharacter, resultingCharacter);
        }

        [Fact]
        public static void StringAndUnicodeSequenceConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;

                var itself = links.Constants.Itself;

                var meaningRoot = links.CreatePoint();
                var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
                var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
                var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
                var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
                var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);

                var powerOf2ToUnaryNumberConverter = new
                    PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
                var addressToUnaryNumberConverter = new
                    AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var charToUnicodeSymbolConverter = new
                    CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                    unicodeSymbolMarker);
```

```csharp
                var unaryNumberToAddressConverter = new
                    UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    powerOf2ToUnaryNumberConverter);
                var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
                var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    frequencyMarker, unaryOne, unaryNumberIncrementer);
                var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                    frequencyPropertyMarker, frequencyMarker);
                var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                    frequencyPropertyOperator, frequencyIncrementer);
                var linkToItsFrequencyNumberConverter = new
                    LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    unaryNumberToAddressConverter);
                var sequenceToItsLocalElementLevelsConverter = new
                    SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    sequenceToItsLocalElementLevelsConverter);

                var stringToUnicodeSymbolConverter = new
                    StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
                    index, optimalVariantConverter, unicodeSequenceMarker);

                var originalString = "Hello";

                var unicodeSequenceLink = stringToUnicodeSymbolConverter.Convert(originalString);

                var unicodeSymbolCriterionMatcher = new
                    UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
                var unicodeSymbolToCharConverter = new
                    UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
                    unicodeSymbolCriterionMatcher);

                var unicodeSequenceCriterionMatcher = new
                    UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);

                var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
                    unicodeSymbolCriterionMatcher.IsMatched);

                var unicodeSequenceToStringConverter = new
                    UnicodeSequenceToStringConverter<ulong>(links,
                    unicodeSequenceCriterionMatcher, sequenceWalker,
                    unicodeSymbolToCharConverter);

                var resultingString =
                    unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);

                Assert.Equal(originalString, resultingString);
            }
        }
    }
}
```

Index