LinksPlatform's Platform.Data.Doublets Class Library

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

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

namespace Platform.Data.Doublets.Decorators
{
    public class LinksCascadeUniquenessAndUsagesResolver<TLink> : LinksUniquenessResolver<TLink>
    {
        public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLink> links) : base(links) { }

        protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
        ↪   newLinkAddress)
        {
            // Use Facade (the last decorator) to ensure recursion working correctly
            Facade.MergeUsages(oldLinkAddress, newLinkAddress);
            return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
        }
    }
}
```

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

```csharp
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Decorators
{
    /// <remarks>
    /// <para>Must be used in conjunction with NonNullContentsLinkDeletionResolver.</para>
    /// <para>Должен использоваться вместе с NonNullContentsLinkDeletionResolver.</para>
    /// </remarks>
    public class LinksCascadeUsagesResolver<TLink> : LinksDecoratorBase<TLink>
    {
        public LinksCascadeUsagesResolver(ILinks<TLink> links) : base(links) { }

        public override void Delete(IList<TLink> restrictions)
        {
            var linkIndex = restrictions[Constants.IndexPart];
            // Use Facade (the last decorator) to ensure recursion working correctly
            Facade.DeleteAllUsages(linkIndex);
            Links.Delete(linkIndex);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
    {
        public LinksConstants<TLink> Constants { get; }

        private ILinks<TLink> _facade;

        public ILinks<TLink> Facade
        {
            get => _facade;
            private set
            {
                _facade = value;
                if (Links is LinksDecoratorBase<TLink> decorator)
                {
                    decorator.Facade = value;
                }
            }
        }

        protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
        {
            Constants = links.Constants;
            Facade = this;
        }

        public virtual TLink Count(IList<TLink> restrictions) => Links.Count(restrictions);
```

```
34
35      public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        ↪  => Links.Each(handler, restrictions);
36
37      public virtual TLink Create(IList<TLink> restrictions) => Links.Create(restrictions);
38
39      public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
        ↪  Links.Update(restrictions, substitution);
40
41      public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);
42   }
43 }
```

./Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs
```
1  using System;
2  using System.Collections.Generic;
3  using Platform.Disposables;
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 LinksDisposableDecoratorBase<TLink> : DisposableBase, ILinks<TLink>
10     {
11         public LinksConstants<TLink> Constants { get; }
12
13         public ILinks<TLink> Links { get; }
14
15         protected LinksDisposableDecoratorBase(ILinks<TLink> links)
16         {
17             Links = links;
18             Constants = links.Constants;
19         }
20
21         public virtual TLink Count(IList<TLink> restrictions) => Links.Count(restrictions);
22
23         public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
           ↪  => Links.Each(handler, restrictions);
24
25         public virtual TLink Create(IList<TLink> restrictions) => Links.Create(restrictions);
26
27         public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
           ↪  Links.Update(restrictions, substitution);
28
29         public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);
30
31         protected override bool AllowMultipleDisposeCalls => true;
32
33         protected override void Dispose(bool manual, bool wasDisposed)
34         {
35             if (!wasDisposed)
36             {
37                 Links.DisposeIfPossible();
38             }
39         }
40     }
41 }
```

./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, IList<TLink> substitution)
```

```
20          {
21              // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
22              Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
23              Links.EnsureInnerReferenceExists(substitution, nameof(substitution));
24              return Links.Update(restrictions, substitution);
25          }
26
27          public override void Delete(IList<TLink> restrictions)
28          {
29              var link = restrictions[Constants.IndexPart];
30              Links.EnsureLinkExists(link, nameof(link));
31              Links.Delete(link);
32          }
33      }
34  }
```

./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, IList<TLink> substitution) =>
            ↪  Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Itself,
            ↪  restrictions, substitution));
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, IList<TLink> substitution)
17          {
```

```
18         var constants = Constants;
19         Links.EnsureCreated(substitution[constants.SourcePart],
   ↪   substitution[constants.TargetPart]);
20         return Links.Update(restrictions, substitution);
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(IList<TLink> restrictions)
12          {
13              var link = Links.Create();
14              return Links.Update(link, link, link);
15          }
16
17          public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
   ↪   Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Null,
   ↪   restrictions, substitution));
18      }
19   }
```

./Platform.Data.Doublets/Decorators/LinksUniquenessResolver.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 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, IList<TLink> substitution)
14          {
15              var newLinkAddress = Links.SearchOrDefault(substitution[Constants.SourcePart],
   ↪   substitution[Constants.TargetPart]);
16              if (_equalityComparer.Equals(newLinkAddress, default))
17              {
18                  return Links.Update(restrictions, substitution);
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                  Facade.Delete(oldLinkAddress);
28              }
29              return newLinkAddress;
30          }
31      }
32   }
```

./Platform.Data.Doublets/Decorators/LinksUniquenessValidator.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 LinksUniquenessValidator<TLink> : LinksDecoratorBase<TLink>
```

```
 8        {
 9            public LinksUniquenessValidator(ILinks<TLink> links) : base(links) { }
10
11            public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
12            {
13                Links.EnsureDoesNotExists(substitution[Constants.SourcePart],
   ↪   substitution[Constants.TargetPart]);
14                return Links.Update(restrictions, substitution);
15            }
16        }
17    }
```

./Platform.Data.Doublets/Decorators/LinksUsagesValidator.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 LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
 8        {
 9            public LinksUsagesValidator(ILinks<TLink> links) : base(links) { }
10
11            public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
12            {
13                Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
14                return Links.Update(restrictions, substitution);
15            }
16
17            public override void Delete(IList<TLink> restrictions)
18            {
19                var link = restrictions[Constants.IndexPart];
20                Links.EnsureNoUsages(link);
21                Links.Delete(link);
22            }
23        }
24    }
```

./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.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 NonNullContentsLinkDeletionResolver<TLink> : LinksDecoratorBase<TLink>
 8        {
 9            public NonNullContentsLinkDeletionResolver(ILinks<TLink> links) : base(links) { }
10
11            public override void Delete(IList<TLink> restrictions)
12            {
13                var linkIndex = restrictions[Constants.IndexPart];
14                Links.EnforceResetValues(linkIndex);
15                Links.Delete(linkIndex);
16            }
17        }
18    }
```

./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 (рассчитываемое по мере использования БД)
19        ///     + меньше объём БД
20        ///     - больше сложность
```

```csharp
    ///
    /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
    ///   поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
    ///   460 752 303 423 488
    /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
    ///   (битовыми строками) - вариант матрицы (выстраеваемой лениво).
    ///
    /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
    ///   выбрасываться только при #if DEBUG
    /// </remarks>
    public class UInt64Links : LinksDisposableDecoratorBase<ulong>
    {
        public UInt64Links(ILinks<ulong> links) : base(links) { }

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

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

        public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
        {
            var constants = Constants;
            var nullConstant = constants.Null;
            if (restrictions.IsNullOrEmpty())
            {
                return nullConstant;
            }
            // TODO: Looks like this is a common type of exceptions linked with restrictions
            //   support
            if (substitution.Count != 3)
            {
                throw new NotSupportedException();
            }
            var indexPartConstant = constants.IndexPart;
            var updatedLink = restrictions[indexPartConstant];
            this.EnsureLinkExists(updatedLink,
                $"{nameof(restrictions)}[{nameof(indexPartConstant)}]");
            var sourcePartConstant = constants.SourcePart;
            var newSource = substitution[sourcePartConstant];
            this.EnsureLinkIsItselfOrExists(newSource,
                $"{nameof(substitution)}[{nameof(sourcePartConstant)}]");
            var targetPartConstant = constants.TargetPart;
            var newTarget = substitution[targetPartConstant];
            this.EnsureLinkIsItselfOrExists(newTarget,
                $"{nameof(substitution)}[{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(IList<ulong> restrictions)
        {
            var linkIndex = restrictions[Constants.IndexPart];
            Links.EnsureLinkExists(linkIndex);
            Links.EnforceResetValues(linkIndex);
```

```
88              this.DeleteAllUsages(linkIndex);
89              Links.Delete(linkIndex);
90          }
91      }
92  }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Platform.Collections;
5  using Platform.Collections.Arrays;
6  using Platform.Collections.Lists;
7  using Platform.Data.Universal;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Decorators
12 {
13     /// <remarks>
14     /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
15     /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed
16     ↪  by itself. But can cause creation (update from nothing) or deletion (update to nothing).
17     /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
18     ↪  DefaultUniLinksBase, that contains logic itself and can be implemented using both
19     ↪  IDoubletLinks and ILinks.)
20     /// </remarks>
    internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
    {
21         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
22
23         public UniLinks(ILinks<TLink> links) : base(links) { }
24
25         private struct Transition
26         {
27             public IList<TLink> Before;
28             public IList<TLink> After;
29
30             public Transition(IList<TLink> before, IList<TLink> after)
31             {
32                 Before = before;
33                 After = after;
34             }
35         }
36
37         //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
38         //public static readonly IReadOnlyList<TLink> NullLink = new
           ↪  ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
           ↪  });
39
40         // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
           ↪  (Links-Expression)
41         public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
           ↪  matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
           ↪  substitutedHandler)
42         {
43             ////List<Transition> transitions = null;
44             ////if (!restriction.IsNullOrEmpty())
45             ////{
46             ////    // Есть причина делать проход (чтение)
47             ////    if (matchedHandler != null)
48             ////    {
49             ////        if (!substitution.IsNullOrEmpty())
50             ////        {
51             ////            // restriction => { 0, 0, 0 } | { 0 } // Create
52             ////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
                ↪  Create / Update
53             ////            // substitution => { 0, 0, 0 } | { 0 } // Delete
54             ////            transitions = new List<Transition>();
55             ////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
56             ////            {
57             ////                // If index is Null, that means we always ignore every other
                ↪  value (they are also Null by definition)
58             ////                var matchDecision = matchedHandler(, NullLink);
59             ////                if (Equals(matchDecision, Constants.Break))
60             ////                    return false;
61             ////                if (!Equals(matchDecision, Constants.Skip))
```

```
 62        ////                            transitions.Add(new Transition(matchedLink, newValue));
 63        ////                }
 64        ////            else
 65        ////            {
 66        ////                Func<T, bool> handler;
 67        ////                handler = link =>
 68        ////                {
 69        ////                    var matchedLink = Memory.GetLinkValue(link);
 70        ////                    var newValue = Memory.GetLinkValue(link);
 71        ////                    newValue[Constants.IndexPart] = Constants.Itself;
 72        ////                    newValue[Constants.SourcePart] =
      ↳  Equals(substitution[Constants.SourcePart], Constants.Itself) ?
      ↳  matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
 73        ////                    newValue[Constants.TargetPart] =
      ↳  Equals(substitution[Constants.TargetPart], Constants.Itself) ?
      ↳  matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
 74        ////                    var matchDecision = matchedHandler(matchedLink, newValue);
 75        ////                    if (Equals(matchDecision, Constants.Break))
 76        ////                        return false;
 77        ////                    if (!Equals(matchDecision, Constants.Skip))
 78        ////                        transitions.Add(new Transition(matchedLink, newValue));
 79        ////                    return true;
 80        ////                };
 81        ////                if (!Memory.Each(handler, restriction))
 82        ////                    return Constants.Break;
 83        ////            }
 84        ////        }
 85        ////        else
 86        ////        {
 87        ////            Func<T, bool> handler = link =>
 88        ////            {
 89        ////                var matchedLink = Memory.GetLinkValue(link);
 90        ////                var matchDecision = matchedHandler(matchedLink, matchedLink);
 91        ////                return !Equals(matchDecision, Constants.Break);
 92        ////            };
 93        ////            if (!Memory.Each(handler, restriction))
 94        ////                return Constants.Break;
 95        ////        }
 96        ////    }
 97        ////    else
 98        ////    {
 99        ////        if (substitution != null)
100        ////        {
101        ////            transitions = new List<IList<T>>();
102        ////            Func<T, bool> handler = link =>
103        ////            {
104        ////                var matchedLink = Memory.GetLinkValue(link);
105        ////                transitions.Add(matchedLink);
106        ////                return true;
107        ////            };
108        ////            if (!Memory.Each(handler, restriction))
109        ////                return Constants.Break;
110        ////        }
111        ////        else
112        ////        {
113        ////            return Constants.Continue;
114        ////        }
115        ////    }
116        ////}
117        ////if (substitution != null)
118        ////{
119        ////    // Есть причина делать замену (запись)
120        ////    if (substitutedHandler != null)
121        ////    {
122        ////    }
123        ////    else
124        ////    {
125        ////    }
126        ////}
127        ////return Constants.Continue;
128
129        //if (restriction.IsNullOrEmpty()) // Create
130        //{
131        //    substitution[Constants.IndexPart] = Memory.AllocateLink();
132        //    Memory.SetLinkValue(substitution);
133        //}
134        //else if (substitution.IsNullOrEmpty()) // Delete
```

```csharp
135             //{
136             //     Memory.FreeLink(restriction[Constants.IndexPart]);
137             //}
138             //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
139             //{
140             //     // No need to collect links to list
141             //     // Skip == Continue
142             //     // No need to check substituedHandler
143             //     if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
    ↪ Constants.Break), restriction))
144             //         return Constants.Break;
145             //}
146             //else // Update
147             //{
148             //     //List<IList<T>> matchedLinks = null;
149             //     if (matchedHandler != null)
150             //     {
151             //         matchedLinks = new List<IList<T>>();
152             //         Func<T, bool> handler = link =>
153             //         {
154             //             var matchedLink = Memory.GetLinkValue(link);
155             //             var matchDecision = matchedHandler(matchedLink);
156             //             if (Equals(matchDecision, Constants.Break))
157             //                 return false;
158             //             if (!Equals(matchDecision, Constants.Skip))
159             //                 matchedLinks.Add(matchedLink);
160             //             return true;
161             //         };
162             //         if (!Memory.Each(handler, restriction))
163             //             return Constants.Break;
164             //     }
165             //     if (!matchedLinks.IsNullOrEmpty())
166             //     {
167             //         var totalMatchedLinks = matchedLinks.Count;
168             //         for (var i = 0; i < totalMatchedLinks; i++)
169             //         {
170             //             var matchedLink = matchedLinks[i];
171             //             if (substitutedHandler != null)
172             //             {
173             //                 var newValue = new List<T>(); // TODO: Prepare value to update here
174             //                 // TODO: Decide is it actually needed to use Before and After
    ↪ substitution handling.
175             //                 var substitutedDecision = substitutedHandler(matchedLink,
    ↪ newValue);
176             //                 if (Equals(substitutedDecision, Constants.Break))
177             //                     return Constants.Break;
178             //                 if (Equals(substitutedDecision, Constants.Continue))
179             //                 {
180             //                     // Actual update here
181             //                     Memory.SetLinkValue(newValue);
182             //                 }
183             //                 if (Equals(substitutedDecision, Constants.Skip))
184             //                 {
185             //                     // Cancel the update. TODO: decide use separate Cancel
    ↪ constant or Skip is enough?
186             //                 }
187             //             }
188             //         }
189             //     }
190             //}
191             return Constants.Continue;
192         }
193
194         public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
    ↪ matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
    ↪ substitutionHandler)
195         {
196             if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
197             {
198                 return Constants.Continue;
199             }
200             else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
    ↪ Check if it is a correct condition
201             {
202                 // Or it only applies to trigger without matchHandler.
203                 throw new NotImplementedException();
204             }
205             else if (!substitution.IsNullOrEmpty()) // Creation
```

```csharp
                    {
                        var before = ArrayPool<TLink>.Empty;
                        // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
                        // (пройти мимо) или пустить (взять)?
                        if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                            Constants.Break))
                        {
                            return Constants.Break;
                        }
                        var after = (IList<TLink>)substitution.ToArray();
                        if (_equalityComparer.Equals(after[0], default))
                        {
                            var newLink = Links.Create();
                            after[0] = newLink;
                        }
                        if (substitution.Count == 1)
                        {
                            after = Links.GetLink(substitution[0]);
                        }
                        else if (substitution.Count == 3)
                        {
                            //Links.Create(after);
                        }
                        else
                        {
                            throw new NotSupportedException();
                        }
                        if (matchHandler != null)
                        {
                            return substitutionHandler(before, after);
                        }
                        return Constants.Continue;
                    }
                    else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
                    {
                        if (patternOrCondition.Count == 1)
                        {
                            var linkToDelete = patternOrCondition[0];
                            var before = Links.GetLink(linkToDelete);
                            if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                                Constants.Break))
                            {
                                return Constants.Break;
                            }
                            var after = ArrayPool<TLink>.Empty;
                            Links.Update(linkToDelete, Constants.Null, Constants.Null);
                            Links.Delete(linkToDelete);
                            if (matchHandler != null)
                            {
                                return substitutionHandler(before, after);
                            }
                            return Constants.Continue;
                        }
                        else
                        {
                            throw new NotSupportedException();
                        }
                    }
                    else // Replace / Update
                    {
                        if (patternOrCondition.Count == 1) //-V3125
                        {
                            var linkToUpdate = patternOrCondition[0];
                            var before = Links.GetLink(linkToUpdate);
                            if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                                Constants.Break))
                            {
                                return Constants.Break;
                            }
                            var after = (IList<TLink>)substitution.ToArray(); //-V3125
                            if (_equalityComparer.Equals(after[0], default))
                            {
                                after[0] = linkToUpdate;
                            }
                            if (substitution.Count == 1)
                            {
                                if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
                                {
```

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

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

./Platform.Data.Doublets/DoubletComparer.cs
```csharp
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets
{
    /// <remarks>
    /// TODO: Может стоит попробовать ref во всех методах (IRefEqualityComparer)
    /// 2x faster with comparer
    /// </remarks>
    public class DoubletComparer<T> : IEqualityComparer<Doublet<T>>
    {
        public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(Doublet<T> x, Doublet<T> y) => x.Equals(y);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int GetHashCode(Doublet<T> obj) => obj.GetHashCode();
    }
}
```

./Platform.Data.Doublets/Doublet.cs

```csharp
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

```csharp
1   using System;
2   using System.Reflection;
3   using System.Reflection.Emit;
4   using Platform.Reflection;
5   using Platform.Converters;
6   using Platform.Exceptions;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets
11  {
12      public class Hybrid<T>
13      {
14          private static readonly Func<object, T> _absAndConvert;
15          private static readonly Func<object, T> _absAndNegateAndConvert;
16
17          static Hybrid()
18          {
19              _absAndConvert = DelegateHelpers.Compile<Func<object, T>>(emiter =>
20              {
21                  Ensure.Always.IsUnsignedInteger<T>();
22                  emiter.LoadArgument(0);
23                  var signedVersion = NumericType<T>.SignedVersion;
24                  var signedVersionField =
                    ↪   typeof(NumericType<T>).GetTypeInfo().GetField("SignedVersion",
                    ↪   BindingFlags.Static | BindingFlags.Public);
25                  //emiter.LoadField(signedVersionField);
26                  emiter.Emit(OpCodes.Ldsfld, signedVersionField);
27                  var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
                    ↪   Types<object, Type>.Array);
28                  emiter.Call(changeTypeMethod);
29                  emiter.UnboxValue(signedVersion);
30                  var absMethod = typeof(Math).GetTypeInfo().GetMethod("Abs", new[] {
                    ↪   signedVersion });
31                  emiter.Call(absMethod);
32                  var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                    ↪   signedVersion });
33                  emiter.Call(unsignedMethod);
34                  emiter.Return();
35              });
36              _absAndNegateAndConvert = DelegateHelpers.Compile<Func<object, T>>(emiter =>
37              {
38                  Ensure.Always.IsUnsignedInteger<T>();
39                  emiter.LoadArgument(0);
```

```csharp
                    var signedVersion = NumericType<T>.SignedVersion;
                    var signedVersionField =
                        typeof(NumericType<T>).GetTypeInfo().GetField("SignedVersion",
                        BindingFlags.Static | BindingFlags.Public);
                    //emiter.LoadField(signedVersionField);
                    emiter.Emit(OpCodes.Ldsfld, signedVersionField);
                    var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
                        Types<object, Type>.Array);
                    emiter.Call(changeTypeMethod);
                    emiter.UnboxValue(signedVersion);
                    var absMethod = typeof(Math).GetTypeInfo().GetMethod("Abs", new[] {
                        signedVersion });
                    emiter.Call(absMethod);
                    var negateMethod = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate
                        ").MakeGenericMethod(signedVersion);
                    emiter.Call(negateMethod);
                    var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                        signedVersion });
                    emiter.Call(unsignedMethod);
                    emiter.Return();
                });
        }

        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.OnDebug.IsUnsignedInteger<T>();
            Value = value;
        }

        public Hybrid(object value) => Value = To.UnsignedAs<T>(Convert.ChangeType(value,
            NumericType<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").MakeGeneric
                Method(signedType);
            //var negate = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate").MakeG
                enericMethod(signedType);
            //var absoluteValue = abs.Invoke(null, new[] { signedValue });
            //var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
                absoluteValue;
            //Value = To.UnsignedAs<T>(resultValue);
            if (isExternal)
            {
                Value = _absAndNegateAndConvert(value);
            }
            else
            {
                Value = _absAndConvert(value);
            }
        }

        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);
```

```csharp
        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);

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

./Platform.Data.Doublets/ILinks.cs

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

using System.Collections.Generic;

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

./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;
using Platform.Data.Doublets.Decorators;

#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());
```

```csharp
                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;
                }
            }
        }

        public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
        ↪  links.Delete(new LinkAddress<TLink>(linkToDelete));

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

        public static TLink First<TLink>(this ILinks<TLink> links)
        {
            TLink firstLink = default;
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (equalityComparer.Equals(links.Count(), default))
            {
                throw new InvalidOperationException("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 InvalidOperationException("В процессе поиска по хранилищу не было
                ↪  найдено связей.");
            }
            return firstLink;
        }

        #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>
```

```csharp
108     public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
    ↪  path)
109     {
110         var current = path[0];
111         //EnsureLinkExists(current, "path");
112         if (!links.Exists(current))
113         {
114             return false;
115         }
116         var equalityComparer = EqualityComparer<TLink>.Default;
117         var constants = links.Constants;
118         for (var i = 1; i < path.Length; i++)
119         {
120             var next = path[i];
121             var values = links.GetLink(current);
122             var source = values[constants.SourcePart];
123             var target = values[constants.TargetPart];
124             if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
    ↪      next))
125             {
126                 //throw new InvalidOperationException(string.Format("Невозможно выбрать
    ↪          путь, так как и Source и Target совпадают с элементом пути {0}.", next));
127                 return false;
128             }
129             if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
    ↪      target))
130             {
131                 //throw new InvalidOperationException(string.Format("Невозможно продолжить
    ↪          путь через элемент пути {0}", next));
132                 return false;
133             }
134             current = next;
135         }
136         return true;
137     }
138
139     /// <remarks>
140     /// Может потребовать дополнительного стека для PathElement's при использовании
    ↪  SequenceWalker.
141     /// </remarks>
142     public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
    ↪  path)
143     {
144         links.EnsureLinkExists(root, "root");
145         var currentLink = root;
146         for (var i = 0; i < path.Length; i++)
147         {
148             currentLink = links.GetLink(currentLink)[path[i]];
149         }
150         return currentLink;
151     }
152
153     public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
    ↪  links, TLink root, ulong size, ulong index)
154     {
155         var constants = links.Constants;
156         var source = constants.SourcePart;
157         var target = constants.TargetPart;
158         if (!Platform.Numbers.Math.IsPowerOfTwo(size))
159         {
160             throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
    ↪          than powers of two are not supported.");
161         }
162         var path = new BitArray(BitConverter.GetBytes(index));
163         var length = Bit.GetLowestPosition(size);
164         links.EnsureLinkExists(root, "root");
165         var currentLink = root;
166         for (var i = length - 1; i >= 0; i--)
167         {
168             currentLink = links.GetLink(currentLink)[path[i] ? target : source];
169         }
170         return currentLink;
171     }
172
173     #endregion
174
175     /// <summary>
176     /// Возвращает индекс указанной связи.
```

```csharp
177         /// </summary>
178         /// <param name="links">Хранилище связей.</param>
179         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪   содержимого.</param>
180         /// <returns>Индекс начальной связи для указанной связи.</returns>
181         [MethodImpl(MethodImplOptions.AggressiveInlining)]
182         public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪   link[links.Constants.IndexPart];

183
184         /// <summary>
185         /// Возвращает индекс начальной (Source) связи для указанной связи.
186         /// </summary>
187         /// <param name="links">Хранилище связей.</param>
188         /// <param name="link">Индекс связи.</param>
189         /// <returns>Индекс начальной связи для указанной связи.</returns>
190         [MethodImpl(MethodImplOptions.AggressiveInlining)]
191         public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪   links.GetLink(link)[links.Constants.SourcePart];

192
193         /// <summary>
194         /// Возвращает индекс начальной (Source) связи для указанной связи.
195         /// </summary>
196         /// <param name="links">Хранилище связей.</param>
197         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪   содержимого.</param>
198         /// <returns>Индекс начальной связи для указанной связи.</returns>
199         [MethodImpl(MethodImplOptions.AggressiveInlining)]
200         public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪   link[links.Constants.SourcePart];

201
202         /// <summary>
203         /// Возвращает индекс конечной (Target) связи для указанной связи.
204         /// </summary>
205         /// <param name="links">Хранилище связей.</param>
206         /// <param name="link">Индекс связи.</param>
207         /// <returns>Индекс конечной связи для указанной связи.</returns>
208         [MethodImpl(MethodImplOptions.AggressiveInlining)]
209         public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪   links.GetLink(link)[links.Constants.TargetPart];

210
211         /// <summary>
212         /// Возвращает индекс конечной (Target) связи для указанной связи.
213         /// </summary>
214         /// <param name="links">Хранилище связей.</param>
215         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪   содержимого.</param>
216         /// <returns>Индекс конечной связи для указанной связи.</returns>
217         [MethodImpl(MethodImplOptions.AggressiveInlining)]
218         public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪   link[links.Constants.TargetPart];

219
220         /// <summary>
221         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪   (handler) для каждой подходящей связи.
222         /// </summary>
223         /// <param name="links">Хранилище связей.</param>
224         /// <param name="handler">Обработчик каждой подходящей связи.</param>
225         /// <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>
226         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪   случае.</returns>
227         [MethodImpl(MethodImplOptions.AggressiveInlining)]
228         public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
            ↪   handler, params TLink[] restrictions)
229             => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
                ↪   links.Constants.Continue);

230
231         /// <summary>
232         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪   (handler) для каждой подходящей связи.
233         /// </summary>
234         /// <param name="links">Хранилище связей.</param>
235         /// <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>
```

```csharp
236         /// <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>
237         /// <param name="handler">Обработчик каждой подходящей связи.</param>
238         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
      ↪    случае.</returns>
239         [MethodImpl(MethodImplOptions.AggressiveInlining)]
240         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
      ↪    Func<TLink, bool> handler)
241         {
242             var constants = links.Constants;
243             return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
      ↪        constants.Break, constants.Any, source, target);
244         }
245
246         /// <summary>
247         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
      ↪    (handler) для каждой подходящей связи.
248         /// </summary>
249         /// <param name="links">Хранилище связей.</param>
250         /// <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>
251         /// <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>
252         /// <param name="handler">Обработчик каждой подходящей связи.</param>
253         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
      ↪    случае.</returns>
254         [MethodImpl(MethodImplOptions.AggressiveInlining)]
255         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
      ↪    Func<IList<TLink>, TLink> handler)
256         {
257             var constants = links.Constants;
258             return links.Each(handler, constants.Any, source, target);
259         }
260
261         [MethodImpl(MethodImplOptions.AggressiveInlining)]
262         public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
      ↪    restrictions)
263         {
264             long arraySize = (Integer<TLink>)links.Count(restrictions);
265             var array = new IList<TLink>[arraySize];
266             if (arraySize > 0)
267             {
268                 var filler = new ArrayFiller<IList<TLink>, TLink>(array,
      ↪            links.Constants.Continue);
269                 links.Each(filler.AddAndReturnConstant, restrictions);
270             }
271             return array;
272         }
273
274         [MethodImpl(MethodImplOptions.AggressiveInlining)]
275         public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
      ↪    restrictions)
276         {
277             long arraySize = (Integer<TLink>)links.Count(restrictions);
278             var array = new TLink[arraySize];
279             if (arraySize > 0)
280             {
281                 var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
282                 links.Each(filler.AddFirstAndReturnConstant, restrictions);
283             }
284             return array;
285         }
286
287         /// <summary>
288         /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
      ↪    в хранилище связей.
289         /// </summary>
290         /// <param name="links">Хранилище связей.</param>
291         /// <param name="source">Начало связи.</param>
292         /// <param name="target">Конец связи.</param>
293         /// <returns>Значение, определяющее существует ли связь.</returns>
294         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
295         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;
296
297         #region Ensure
298         // TODO: May be move to EnsureExtensions or make it both there and here
299
300         [MethodImpl(MethodImplOptions.AggressiveInlining)]
301         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
                reference, string argumentName)
302         {
303             if (links.Constants.IsInnerReference(reference) && !links.Exists(reference))
304             {
305                 throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
306             }
307         }
308
309         [MethodImpl(MethodImplOptions.AggressiveInlining)]
310         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
                IList<TLink> restrictions, string argumentName)
311         {
312             for (int i = 0; i < restrictions.Count; i++)
313             {
314                 links.EnsureInnerReferenceExists(restrictions[i], argumentName);
315             }
316         }
317
318         [MethodImpl(MethodImplOptions.AggressiveInlining)]
319         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
                restrictions)
320         {
321             for (int i = 0; i < restrictions.Count; i++)
322             {
323                 links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
324             }
325         }
326
327         [MethodImpl(MethodImplOptions.AggressiveInlining)]
328         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
                string argumentName)
329         {
330             var equalityComparer = EqualityComparer<TLink>.Default;
331             if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
332             {
333                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
334             }
335         }
336
337         [MethodImpl(MethodImplOptions.AggressiveInlining)]
338         public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
                link, string argumentName)
339         {
340             var equalityComparer = EqualityComparer<TLink>.Default;
341             if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
342             {
343                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
344             }
345         }
346
347         /// <param name="links">Хранилище связей.</param>
348         [MethodImpl(MethodImplOptions.AggressiveInlining)]
349         public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
                TLink target)
350         {
351             if (links.Exists(source, target))
352             {
353                 throw new LinkWithSameValueAlreadyExistsException();
354             }
355         }
356
357         /// <param name="links">Хранилище связей.</param>
358         public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
359         {
360             if (links.HasUsages(link))
361             {
362                 throw new ArgumentLinkHasDependenciesException<TLink>(link);
363             }
364         }
```

```csharp
365
366        /// <param name="links">Хранилище связей.</param>
367        public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
     ↪  addresses) => links.EnsureCreated(links.Create, addresses);
368
369        /// <param name="links">Хранилище связей.</param>
370        public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
     ↪  addresses) => links.EnsureCreated(links.CreatePoint, addresses);
371
372        /// <param name="links">Хранилище связей.</param>
373        public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
     ↪  params TLink[] addresses)
374        {
375            var constants = links.Constants;
376
377            var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
     ↪  !links.Exists(x)));
378            if (nonExistentAddresses.Count > 0)
379            {
380                var max = nonExistentAddresses.Max();
381                max = (Integer<TLink>)System.Math.Min((ulong)(Integer<TLink>)max,
     ↪  (ulong)(Integer<TLink>)constants.PossibleInnerReferencesRange.Maximum);
382                var createdLinks = new List<TLink>();
383                var equalityComparer = EqualityComparer<TLink>.Default;
384                TLink createdLink = creator();
385                while (!equalityComparer.Equals(createdLink, max))
386                {
387                    createdLinks.Add(createdLink);
388                }
389                for (var i = 0; i < createdLinks.Count; i++)
390                {
391                    if (!nonExistentAddresses.Contains(createdLinks[i]))
392                    {
393                        links.Delete(createdLinks[i]);
394                    }
395                }
396            }
397        }
398
399        #endregion
400
401        /// <param name="links">Хранилище связей.</param>
402        public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
403        {
404            var constants = links.Constants;
405            var values = links.GetLink(link);
406            TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
     ↪  constants.Any));
407            var equalityComparer = EqualityComparer<TLink>.Default;
408            if (equalityComparer.Equals(values[constants.SourcePart], link))
409            {
410                usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
411            }
412            TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
     ↪  link));
413            if (equalityComparer.Equals(values[constants.TargetPart], link))
414            {
415                usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
416            }
417            return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
418        }
419
420        /// <param name="links">Хранилище связей.</param>
421        [MethodImpl(MethodImplOptions.AggressiveInlining)]
422        public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
     ↪  Comparer<TLink>.Default.Compare(links.CountUsages(link), Integer<TLink>.Zero) > 0;
423
424        /// <param name="links">Хранилище связей.</param>
425        [MethodImpl(MethodImplOptions.AggressiveInlining)]
426        public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
     ↪  TLink target)
427        {
428            var constants = links.Constants;
429            var values = links.GetLink(link);
430            var equalityComparer = EqualityComparer<TLink>.Default;
431            return equalityComparer.Equals(values[constants.SourcePart], source) &&
     ↪  equalityComparer.Equals(values[constants.TargetPart], target);
432        }
```

```
433
434        /// <summary>
435        /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
436        /// </summary>
437        /// <param name="links">Хранилище связей.</param>
438        /// <param name="source">Индекс связи, которая является началом для искомой
           ↪  связи.</param>
439        /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
440        /// <returns>Индекс искомой связи с указанными Source (началом) и Target
           ↪  (концом).</returns>
441        [MethodImpl(MethodImplOptions.AggressiveInlining)]
442        public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
           ↪  target)
443        {
444            var contants = links.Constants;
445            var setter = new Setter<TLink, TLink>(contants.Continue, contants.Break, default);
446            links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
447            return setter.Result;
448        }
449
450        /// <param name="links">Хранилище связей.</param>
451        [MethodImpl(MethodImplOptions.AggressiveInlining)]
452        public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
453
454        /// <param name="links">Хранилище связей.</param>
455        [MethodImpl(MethodImplOptions.AggressiveInlining)]
456        public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
457        {
458            var link = links.Create();
459            return links.Update(link, link, link);
460        }
461
462        /// <param name="links">Хранилище связей.</param>
463        [MethodImpl(MethodImplOptions.AggressiveInlining)]
464        public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
           ↪  target) => links.Update(links.Create(), source, target);
465
466        /// <summary>
467        /// Обновляет связь с указанными началом (Source) и концом (Target)
468        /// на связь с указанными началом (NewSource) и концом (NewTarget).
469        /// </summary>
470        /// <param name="links">Хранилище связей.</param>
471        /// <param name="link">Индекс обновляемой связи.</param>
472        /// <param name="newSource">Индекс связи, которая является началом связи, на которую
           ↪  выполняется обновление.</param>
473        /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
           ↪  выполняется обновление.</param>
474        /// <returns>Индекс обновлённой связи.</returns>
475        [MethodImpl(MethodImplOptions.AggressiveInlining)]
476        public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
           ↪  TLink newTarget) => links.Update(new LinkAddress<TLink>(link), new Link<TLink>(link,
           ↪  newSource, newTarget));
477
478        /// <summary>
479        /// Обновляет связь с указанными началом (Source) и концом (Target)
480        /// на связь с указанными началом (NewSource) и концом (NewTarget).
481        /// </summary>
482        /// <param name="links">Хранилище связей.</param>
483        /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
           ↪  может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
           ↪  Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
           ↪  связи.</param>
484        /// <returns>Индекс обновлённой связи.</returns>
485        [MethodImpl(MethodImplOptions.AggressiveInlining)]
486        public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
487        {
488            if (restrictions.Length == 2)
489            {
490                return links.MergeAndDelete(restrictions[0], restrictions[1]);
491            }
492            if (restrictions.Length == 4)
493            {
494                return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
                   ↪  restrictions[2], restrictions[3]);
495            }
496            else
497            {
```

```csharp
498                 return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
499             }
500         }
501
502         [MethodImpl(MethodImplOptions.AggressiveInlining)]
503         public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
            ↪ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
504         {
505             var equalityComparer = EqualityComparer<TLink>.Default;
506             var constants = links.Constants;
507             var restrictionsIndex = restrictions[constants.IndexPart];
508             var substitutionIndex = substitution[constants.IndexPart];
509             if (equalityComparer.Equals(substitutionIndex, default))
510             {
511                 substitutionIndex = restrictionsIndex;
512             }
513             var source = substitution[constants.SourcePart];
514             var target = substitution[constants.TargetPart];
515             source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
516             target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
517             return new Link<TLink>(substitutionIndex, source, target);
518         }
519
520         /// <summary>
521         /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
            ↪ с указанными Source (началом) и Target (концом).
522         /// </summary>
523         /// <param name="links">Хранилище связей.</param>
524         /// <param name="source">Индекс связи, которая является началом на создаваемой
            ↪ связи.</param>
525         /// <param name="target">Индекс связи, которая является концом для создаваемой
            ↪ связи.</param>
526         /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
527         [MethodImpl(MethodImplOptions.AggressiveInlining)]
528         public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
529         {
530             var link = links.SearchOrDefault(source, target);
531             if (EqualityComparer<TLink>.Default.Equals(link, default))
532             {
533                 link = links.CreateAndUpdate(source, target);
534             }
535             return link;
536         }
537
538         /// <summary>
539         /// Обновляет связь с указанными началом (Source) и концом (Target)
540         /// на связь с указанными началом (NewSource) и концом (NewTarget).
541         /// </summary>
542         /// <param name="links">Хранилище связей.</param>
543         /// <param name="source">Индекс связи, которая является началом обновляемой
            ↪ связи.</param>
544         /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
545         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪ выполняется обновление.</param>
546         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪ выполняется обновление.</param>
547         /// <returns>Индекс обновлённой связи.</returns>
548         [MethodImpl(MethodImplOptions.AggressiveInlining)]
549         public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
            ↪ TLink target, TLink newSource, TLink newTarget)
550         {
551             var equalityComparer = EqualityComparer<TLink>.Default;
552             var link = links.SearchOrDefault(source, target);
553             if (equalityComparer.Equals(link, default))
554             {
555                 return links.CreateAndUpdate(newSource, newTarget);
556             }
557             if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
            ↪ target))
558             {
559                 return link;
560             }
561             return links.Update(link, newSource, newTarget);
562         }
563
564         /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
```

```csharp
        /// <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)
        {
            var anyConstant = links.Constants.Any;
            var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
            links.DeleteByQuery(usagesAsSourceQuery);
            var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
            links.DeleteByQuery(usagesAsTargetQuery);
        }

        public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
        {
            var count = (Integer<TLink>)links.Count(query);
            if (count > 0)
            {
                var queryResult = new TLink[count];
                var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
                ↪  links.Constants.Continue);
                links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
                for (var i = (long)count - 1; i >= 0; i--)
                {
                    links.Delete(queryResult[i]);
                }
            }
        }

        // TODO: Move to Platform.Data
        public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
            var nullConstant = links.Constants.Null;
            var equalityComparer = EqualityComparer<TLink>.Default;
            var link = links.GetLink(linkIndex);
            for (int i = 1; i < link.Count; i++)
            {
                if (!equalityComparer.Equals(link[i], nullConstant))
                {
                    return false;
                }
            }
            return true;
        }

        // TODO: Create a universal version of this method in Platform.Data (with using of for
        ↪  loop)
        public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
            var nullConstant = links.Constants.Null;
            var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
```

```csharp
638                 links.Update(updateRequest);
639             }
640
641             // TODO: Create a universal version of this method in Platform.Data (with using of for
           ↪  loop)
642             public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
643             {
644                 if (!links.AreValuesReset(linkIndex))
645                 {
646                     links.ResetValues(linkIndex);
647                 }
648             }
649
650             /// <summary>
651             /// Merging two usages graphs, all children of old link moved to be children of new link
           ↪  or deleted.
652             /// </summary>
653             public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
           ↪  TLink newLinkIndex)
654             {
655                 var equalityComparer = EqualityComparer<TLink>.Default;
656                 if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
657                 {
658                     var constants = links.Constants;
659                     var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
               ↪  constants.Any);
660                     long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
661                     var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
               ↪  oldLinkIndex);
662                     long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
663                     var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
               ↪  usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
664                     if (!isStandalonePoint)
665                     {
666                         var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
667                         if (totalUsages > 0)
668                         {
669                             var usages = ArrayPool.Allocate<TLink>(totalUsages);
670                             var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
                   ↪  links.Constants.Continue);
671                             var i = 0L;
672                             if (usagesAsSourceCount > 0)
673                             {
674                                 links.Each(usagesFiller.AddFirstAndReturnConstant,
                       ↪  usagesAsSourceQuery);
675                                 for (; i < usagesAsSourceCount; i++)
676                                 {
677                                     var usage = usages[i];
678                                     if (!equalityComparer.Equals(usage, oldLinkIndex))
679                                     {
680                                         links.Update(usage, newLinkIndex, links.GetTarget(usage));
681                                     }
682                                 }
683                             }
684                             if (usagesAsTargetCount > 0)
685                             {
686                                 links.Each(usagesFiller.AddFirstAndReturnConstant,
                       ↪  usagesAsTargetQuery);
687                                 for (; i < usages.Length; i++)
688                                 {
689                                     var usage = usages[i];
690                                     if (!equalityComparer.Equals(usage, oldLinkIndex))
691                                     {
692                                         links.Update(usage, links.GetSource(usage), newLinkIndex);
693                                     }
694                                 }
695                             }
696                             ArrayPool.Free(usages);
697                         }
698                     }
699                 }
700                 return newLinkIndex;
701             }
702
703             /// <summary>
704             /// Replace one link with another (replaced link is deleted, children are updated or
           ↪  deleted).
```

```
705        /// </summary>
706        [MethodImpl(MethodImplOptions.AggressiveInlining)]
707        public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
           ↪  TLink newLinkIndex)
708        {
709            var equalityComparer = EqualityComparer<TLink>.Default;
710            if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
711            {
712                links.MergeUsages(oldLinkIndex, newLinkIndex);
713                links.Delete(oldLinkIndex);
714            }
715            return newLinkIndex;
716        }
717
718        public static ILinks<TLink>
           ↪  DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
719        {
720            links = new LinksCascadeUsagesResolver<TLink>(links);
721            links = new NonNullContentsLinkDeletionResolver<TLink>(links);
722            links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
723            return links;
724        }
725    }
726 }
```

./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.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 FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
9       {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
11
12          private readonly TLink _frequencyMarker;
13          private readonly TLink _unaryOne;
14          private readonly IIncrementer<TLink> _unaryNumberIncrementer;
15
16          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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets
4  {
5      public interface ISynchronizedLinks<TLink> : ISynchronizedLinks<TLink, ILinks<TLink>,
   ↪  LinksConstants<TLink>>, ILinks<TLink>
6      {
7      }
8  }
```

./Platform.Data.Doublets/Link.cs
```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Platform.Exceptions;
5  using Platform.Ranges;
6  using Platform.Singletons;
7  using Platform.Collections.Lists;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets
12 {
13     /// <summary>
14     /// Структура описывающая уникальную связь.
15     /// </summary>
16     public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
17     {
18         public static readonly Link<TLink> Null = new Link<TLink>();
19
20         private static readonly LinksConstants<TLink> _constants =
   ↪  Default<LinksConstants<TLink>>.Instance;
21         private static readonly EqualityComparer<TLink> _equalityComparer =
   ↪  EqualityComparer<TLink>.Default;
22
23         private const int Length = 3;
24
25         public readonly TLink Index;
26         public readonly TLink Source;
27         public readonly TLink Target;
28
29         public Link(params TLink[] values)
30         {
31             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
   ↪  _constants.Null;
32             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
   ↪  _constants.Null;
33             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
   ↪  _constants.Null;
34         }
35
36         public Link(IList<TLink> values)
37         {
38             Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
   ↪  _constants.Null;
```

```csharp
                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();

        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.OnDebug.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();
        }
```

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

111        public IEnumerator<TLink> GetEnumerator()
112        {
113            yield return Index;
114            yield return Source;
115            yield return Target;
116        }

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

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

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

124        public void CopyTo(TLink[] array, int arrayIndex)
125        {
126            Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
127            Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                ↪  nameof(arrayIndex));
128            if (arrayIndex + Length > array.Length)
129            {
130                throw new InvalidOperationException();
131            }
132            array[arrayIndex++] = Index;
133            array[arrayIndex++] = Source;
134            array[arrayIndex] = Target;
135        }

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

139        public int IndexOf(TLink item)
140        {
141            if (_equalityComparer.Equals(Index, item))
142            {
143                return _constants.IndexPart;
144            }
145            if (_equalityComparer.Equals(Source, item))
146            {
147                return _constants.SourcePart;
148            }
149            if (_equalityComparer.Equals(Target, item))
150            {
151                return _constants.TargetPart;
152            }
153            return -1;
154        }

156        public void Insert(int index, TLink item) => throw new NotSupportedException();

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

160        #endregion
161    }
162 }
```

./Platform.Data.Doublets/LinkExtensions.cs

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

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

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 number)
19            {
20                var nullConstant = Links.Constants.Null;
21                var one = Integer<TLink>.One;
22                var target = nullConstant;
23                for (int i = 0; !_equalityComparer.Equals(number, default) && i <
                  ↪  NumericType<TLink>.BitsLength; i++)
24                {
25                    if (_equalityComparer.Equals(Bit.And(number, one), one))
26                    {
27                        target = _equalityComparer.Equals(target, nullConstant)
28                            ? _powerOf2ToUnaryNumberConverter.Convert(i)
29                            : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
30                    }
31                    number = Bit.ShiftRight(number, 1);
32                }
33                return target;
34            }
35        }
36    }
```

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

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

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

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

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

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

        public TLink Convert(Doublet<TLink> doublet)
        {
            var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
            if (_equalityComparer.Equals(link, default))
            {
                throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
            }
            var frequency = _frequencyPropertyOperator.Get(link);
            if (_equalityComparer.Equals(frequency, default))
            {
                return default;
            }
            var frequencyNumber = Links.GetSource(frequency);
            return _unaryNumberToAddressConverter.Convert(frequencyNumber);
        }
    }
}
```

./Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs

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

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
        IConverter<int, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;

        private readonly TLink[] _unaryNumberPowersOf2;

        public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
        {
            _unaryNumberPowersOf2 = new TLink[64];
            _unaryNumberPowersOf2[0] = one;
        }

        public TLink Convert(int power)
        {
            Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
                - 1), nameof(power));
            if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
            {
                return _unaryNumberPowersOf2[power];
            }
            var previousPowerOf2 = Convert(power - 1);
            var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
            _unaryNumberPowersOf2[power] = powerOf2;
            return powerOf2;
        }
    }
}
```

```csharp
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Numbers;

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

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

        private Dictionary<TLink, TLink> _unaryToUInt64;
        private readonly TLink _unaryOne;

        public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
            : base(links)
        {
            _unaryOne = unaryOne;
            InitUnaryToUInt64();
        }

        private void InitUnaryToUInt64()
        {
            var one = Integer<TLink>.One;
            _unaryToUInt64 = new Dictionary<TLink, TLink>
            {
                { _unaryOne, one }
            };
            var unary = _unaryOne;
            var number = one;
            for (var i = 1; i < 64; i++)
            {
                unary = Links.GetOrCreate(unary, unary);
                number = Double(number);
                _unaryToUInt64.Add(unary, number);
            }
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink Double(TLink number) => (Integer<TLink>)((Integer<TLink>)number *
            2UL);
    }
}
```

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Interfaces;
4   using Platform.Reflection;
5   using Platform.Numbers;
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 < NumericType<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 =
            ↪  Bit.Or(target, Bit.ShiftLeft(Integer<TLink>.One, powerOf2Index));
54      }
55  }
```

```csharp
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;
14
15         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
          ↪  propertyValueMarker) : base(links)
16         {
17             _propertyMarker = propertyMarker;
18             _propertyValueMarker = propertyValueMarker;
19         }
20
21         public TLink Get(TLink link)
22         {
23             var property = Links.SearchOrDefault(link, _propertyMarker);
24             var container = GetContainer(property);
25             var value = GetValue(container);
26             return value;
27         }
28
29         private TLink GetContainer(TLink property)
30         {
31             var valueContainer = default(TLink);
32             if (_equalityComparer.Equals(property, default))
33             {
34                 return valueContainer;
35             }
36             var constants = Links.Constants;
37             var countinueConstant = constants.Continue;
38             var breakConstant = constants.Break;
39             var anyConstant = constants.Any;
40             var query = new Link<TLink>(anyConstant, property, anyConstant);
41             Links.Each(candidate =>
42             {
43                 var candidateTarget = Links.GetTarget(candidate);
44                 var valueTarget = Links.GetTarget(candidateTarget);
45                 if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
46                 {
47                     valueContainer = Links.GetIndex(candidate);
48                     return breakConstant;
49                 }
50                 return countinueConstant;
51             }, query);
52             return valueContainer;
53         }
54
```

```csharp
            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/ILinksListMethods.cs

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public interface ILinksListMethods<TLink>
    {
        void Detach(TLink freeLink);
        void AttachAsFirst(TLink link);
    }
}
```

./Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.cs

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

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public interface ILinksTreeMethods<TLink>
    {
        TLink CountUsages(TLink link);
        TLink Search(TLink source, TLink target);
        TLink EachUsage(TLink source, Func<IList<TLink>, TLink> handler);
        void Detach(ref TLink firstAsSource, TLink linkIndex);
        void Attach(ref TLink firstAsSource, TLink linkIndex);
    }
}
```

./Platform.Data.Doublets/ResizableDirectMemory/LinksAVLBalancedTreeMethodsBase.cs

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

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public unsafe abstract class LinksAVLBalancedTreeMethodsBase<TLink> :
↪    SizedAndThreadedAVLBalancedTreeMethods<TLink>
    {
        private readonly ResizableDirectMemoryLinks<TLink> _memory;
        private readonly LinksConstants<TLink> _constants;
        protected readonly byte* Links;
        protected readonly byte* Header;

        public LinksAVLBalancedTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory, byte*
↪    links, byte* header)
        {
            Links = links;
            Header = header;
            _memory = memory;
            _constants = memory.Constants;
        }
```

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
        ↪   rootSource, TLink rootTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
        ↪   rootSource, TLink rootTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        internal virtual ref LinksHeader<TLink> GetHeaderReference() => ref
        ↪   AsRef<LinksHeader<TLink>>(Header);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        internal virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
        ↪   AsRef<RawLink<TLink>>((void*)(Links + RawLink<TLink>.SizeInBytes *
        ↪   (Integer<TLink>)link));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual Link<TLink> GetLinkValues(TLink current) =>
        ↪   _memory.GetLinkStruct(current);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
        {
            ref var firstLink = ref GetLinkReference(first);
            ref var secondLink = ref GetLinkReference(second);
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
            ↪   secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
        {
            ref var firstLink = ref GetLinkReference(first);
            ref var secondLink = ref GetLinkReference(second);
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
            ↪   secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5, -5);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
        ↪   Bit<TLink>.PartialWrite(storedValue, size, 5, -5);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected bool GetLeftIsChildValue(TLink value)
        {
            unchecked
            {
                //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
                return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static void SetLeftIsChildValue(ref TLink storedValue, bool value)
        {
            unchecked
            {
                var previousValue = storedValue;
                var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
                ↪   1);
                storedValue = modified;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected bool GetRightIsChildValue(TLink value)
        {
            unchecked
```

```csharp
            {
                //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
                return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static void SetRightIsChildValue(ref TLink storedValue, bool value)
        {
            unchecked
            {
                var previousValue = storedValue;
                var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
                → 1);
                storedValue = modified;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static sbyte GetBalanceValue(TLink storedValue)
        {
            unchecked
            {
                var value = (int)(Integer<TLink>)Bit<TLink>.PartialRead(storedValue, 0, 3);
                value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
                → end of sbyte
                return (sbyte)value;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static void SetBalanceValue(ref TLink storedValue, sbyte value)
        {
            unchecked
            {
                var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
                → 3);
                var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
                storedValue = modified;
            }
        }

        public TLink this[TLink index]
        {
            get
            {
                var root = GetTreeRoot();
                if (GreaterOrEqualThan(index, GetSize(root)))
                {
                    return Zero;
                }
                while (!EqualToZero(root))
                {
                    var left = GetLeftOrDefault(root);
                    var leftSize = GetSizeOrZero(left);
                    if (LessThan(index, leftSize))
                    {
                        root = left;
                        continue;
                    }
                    if (IsEquals(index, leftSize))
                    {
                        return root;
                    }
                    root = GetRightOrDefault(root);
                    index = Subtract(index, Increment(leftSize));
                }
                return Zero; // TODO: Impossible situation exception (only if tree structure
                → broken)
            }
        }

        /// <summary>
        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
        → (концом).
        /// </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>
```

```csharp
        public TLink Search(TLink source, TLink target)
        {
            var root = GetTreeRoot();
            while (!EqualToZero(root))
            {
                ref var rootLink = ref GetLinkReference(root);
                var rootSource = rootLink.Source;
                var rootTarget = rootLink.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 Zero;
        }

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

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

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

```
247                            first = current;
248                        }
249                        current = GetLeftOrDefault(current);
250                    }
251                    else
252                    {
253                        current = GetRightOrDefault(current);
254                    }
255                }
256                if (!EqualToZero(first))
257                {
258                    current = first;
259                    while (true)
260                    {
261                        if (IsEquals(handler(GetLinkValues(current)), _constants.Break))
262                        {
263                            return _constants.Break;
264                        }
265                        current = GetNext(current);
266                        if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
267                        {
268                            break;
269                        }
270                    }
271                }
272                return _constants.Continue;
273            }
274
275            protected override void PrintNodeValue(TLink node, StringBuilder sb)
276            {
277                ref var link = ref GetLinkReference(node);
278                sb.Append(' ');
279                sb.Append(link.Source);
280                sb.Append('-');
281                sb.Append('>');
282                sb.Append(link.Target);
283            }
284        }
285    }
```

./Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs

```
1    using Platform.Unsafe;
2
3    namespace Platform.Data.Doublets.ResizableDirectMemory
4    {
5        internal struct LinksHeader<TLink>
6        {
7            public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;
8
9            public TLink AllocatedLinks;
10           public TLink ReservedLinks;
11           public TLink FreeLinks;
12           public TLink FirstFreeLink;
13           public TLink FirstAsSource;
14           public TLink FirstAsTarget;
15           public TLink LastFreeLink;
16           public TLink Reserved8;
17        }
18    }
```

./Platform.Data.Doublets/ResizableDirectMemory/LinksSourcesAVLBalancedTreeMethods.cs

```
1    using System.Runtime.CompilerServices;
2    using Platform.Numbers;
3    using static System.Runtime.CompilerServices.Unsafe;
4
5    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7    namespace Platform.Data.Doublets.ResizableDirectMemory
8    {
9        public unsafe class LinksSourcesAVLBalancedTreeMethods<TLink> :
           LinksAVLBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
10       {
11           public LinksSourcesAVLBalancedTreeMethods(ResizableDirectMemoryLinks<TLink> memory,
              byte* links, byte* header) : base(memory, links, header) { }
12
13           [MethodImpl(MethodImplOptions.AggressiveInlining)]
14           protected unsafe override ref TLink GetLeftReference(TLink node) => ref
              GetLinkReference(node).LeftAsSource;
15
16           [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        protected unsafe override ref TLink GetRightReference(TLink node) => ref
        ↪   GetLinkReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
        ↪   GetLinkReference(node).LeftAsSource = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
        ↪   GetLinkReference(node).RightAsSource = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) =>
        ↪   GetSizeValue(GetLinkReference(node).SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
        ↪   GetLinkReference(node).SizeAsSource, size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChild(TLink node) =>
        ↪   GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChild(TLink node, bool value) =>
        ↪   SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChild(TLink node) =>
        ↪   GetRightIsChildValue(GetLinkReference(node).SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChild(TLink node, bool value) =>
        ↪   SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalance(TLink node) =>
        ↪   GetBalanceValue(GetLinkReference(node).SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
        ↪   GetLinkReference(node).SizeAsSource, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪   TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
        ↪   (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

```csharp
using System.Runtime.CompilerServices;
using Platform.Numbers;
using static System.Runtime.CompilerServices.Unsafe;

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public unsafe class LinksTargetsAVLBalancedTreeMethods<TLink> :
        LinksAVLBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
    {
        public LinksTargetsAVLBalancedTreeMethods(ResizableDirectMemoryLinks<TLink> memory,
            byte* links, byte* header) : base(memory, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected unsafe override ref TLink GetLeftReference(TLink node) => ref
            GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected unsafe override ref TLink GetRightReference(TLink node) => ref
            GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
            GetLinkReference(node).LeftAsTarget = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
            GetLinkReference(node).RightAsTarget = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) =>
            GetSizeValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
            GetLinkReference(node).SizeAsTarget, size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChild(TLink node) =>
            GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChild(TLink node, bool value) =>
            SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChild(TLink node) =>
            GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChild(TLink node, bool value) =>
            SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalance(TLink node) =>
            GetBalanceValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
            GetLinkReference(node).SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
 62            protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
        ↪  (IsEquals(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
 63
 64            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 65            protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
        ↪  (IsEquals(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
 66
 67            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 68            protected override void ClearNode(TLink node)
 69            {
 70                ref var link = ref GetLinkReference(node);
 71                link.LeftAsTarget = Zero;
 72                link.RightAsTarget = Zero;
 73                link.SizeAsTarget = Zero;
 74            }
 75        }
 76    }
```

./Platform.Data.Doublets/ResizableDirectMemory/RawLink.cs

```csharp
 1    using Platform.Unsafe;
 2
 3    namespace Platform.Data.Doublets.ResizableDirectMemory
 4    {
 5        internal struct RawLink<TLink>
 6        {
 7            public static readonly long SizeInBytes = Structure<RawLink<TLink>>.Size;
 8
 9            public TLink Source;
10            public TLink Target;
11            public TLink LeftAsSource;
12            public TLink RightAsSource;
13            public TLink SizeAsSource;
14            public TLink LeftAsTarget;
15            public TLink RightAsTarget;
16            public TLink SizeAsTarget;
17        }
18    }
```

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

```csharp
 1    using System;
 2    using System.Collections.Generic;
 3    using System.Runtime.CompilerServices;
 4    using Platform.Disposables;
 5    using Platform.Singletons;
 6    using Platform.Collections.Arrays;
 7    using Platform.Numbers;
 8    using Platform.Memory;
 9    using Platform.Data.Exceptions;
10    using static Platform.Numbers.Arithmetic;
11    using static System.Runtime.CompilerServices.Unsafe;
12
13    #pragma warning disable 0649
14    #pragma warning disable 169
15    #pragma warning disable 618
16    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
17
18    // ReSharper disable StaticMemberInGenericType
19    // ReSharper disable BuiltInTypeReferenceStyle
20    // ReSharper disable MemberCanBePrivate.Local
21    // ReSharper disable UnusedMember.Local
22
23    namespace Platform.Data.Doublets.ResizableDirectMemory
24    {
25        public unsafe partial class ResizableDirectMemoryLinks<TLink> : DisposableBase, ILinks<TLink>
26        {
27            private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
28            private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
29
30            /// <summary>Возвращает размер одной связи в 6айтах.</summary>
31            public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
32
33            public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
34
35            public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
36
37            private readonly long _memoryReservationStep;
38
39            private readonly IResizableDirectMemory _memory;
```

```csharp
        private byte* _header;
        private byte* _links;

        private ILinksTreeMethods<TLink> _targetsTreeMethods;
        private ILinksTreeMethods<TLink> _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 ILinksListMethods<TLink> _unusedLinksListMethods;

        /// <summary>
        /// Возвращает общее число связей находящихся в хранилище.
        /// </summary>
        private TLink Total
        {
            get
            {
                ref var header = ref AsRef<LinksHeader<TLink>>(_header);
                return Subtract(header.AllocatedLinks, header.FreeLinks);
            }
        }

        public LinksConstants<TLink> Constants { get; }

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

        /// <summary>
        /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
        ///   минимальным шагом расширения базы данных.
        /// </summary>
        /// <param name="address">Полный путь к файлу базы данных.</param>
        /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
        ///   байтах.</param>
        public ResizableDirectMemoryLinks(string address, long memoryReservationStep) : this(new
            FileMappedResizableDirectMemory(address, memoryReservationStep),
            memoryReservationStep) { }

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

        public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            memoryReservationStep)
        {
            Constants = Default<LinksConstants<TLink>>.Instance;
            _memory = memory;
            _memoryReservationStep = memoryReservationStep;
            if (memory.ReservedCapacity < memoryReservationStep)
            {
                memory.ReservedCapacity = memoryReservationStep;
            }
            SetPointers(_memory);
            ref var header = ref AsRef<LinksHeader<TLink>>(_header);
            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
            _memory.UsedCapacity = ((Integer<TLink>)header.AllocatedLinks * LinkSizeInBytes) +
                LinkHeaderSizeInBytes;
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            header.ReservedLinks = (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;
            }
```

```csharp
            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;
                    }
                    ref var storedLinkValue = ref GetLinkUnsafe(index);
                    if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
                        _equalityComparer.Equals(storedLinkValue.Target, 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);
                    }
                    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;
                    }
                    ref var storedLinkValue = ref GetLinkUnsafe(index);
                    if (!_equalityComparer.Equals(source, Constants.Any) &&
                        !_equalityComparer.Equals(target, Constants.Any))
                    {
                        if (_equalityComparer.Equals(storedLinkValue.Source, source) &&
                            _equalityComparer.Equals(storedLinkValue.Target, target))
```

```csharp
                    {
                        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(storedLinkValue.Source, value) ||
                    _equalityComparer.Equals(storedLinkValue.Target, 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>)AsRef<LinksHeader<TLink>>(_header).AllocatedLinks) <= 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))
            {
                return Constants.Continue;
            }
            return handler(GetLinkStruct(index));
        }
        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 Each(handler, ArrayPool<TLink>.Empty);
                }
                if (_equalityComparer.Equals(Each(handler, new[] { index, value,
                ↪ Constants.Any }), Constants.Break))
                {
                    return Constants.Break;
                }
                return Each(handler, new[] { index, Constants.Any, value });
            }
            else
            {
                if (!Exists(index))
                {
                    return Constants.Continue;
```

```
254                        }
255                        if (_equalityComparer.Equals(value, Constants.Any))
256                        {
257                            return handler(GetLinkStruct(index));
258                        }
259                        ref var storedLinkValue = ref GetLinkUnsafe(index);
260                        if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
261                            _equalityComparer.Equals(storedLinkValue.Target, value))
262                        {
263                            return handler(GetLinkStruct(index));
264                        }
265                        return Constants.Continue;
266                    }
267                }
268            if (restrictions.Count == 3)
269            {
270                var index = restrictions[Constants.IndexPart];
271                var source = restrictions[Constants.SourcePart];
272                var target = restrictions[Constants.TargetPart];
273                if (_equalityComparer.Equals(index, Constants.Any))
274                {
275                    if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ _equalityComparer.Equals(target, Constants.Any))
276                    {
277                        return Each(handler, ArrayPool<TLink>.Empty);
278                    }
279                    else if (_equalityComparer.Equals(source, Constants.Any))
280                    {
281                        return _targetsTreeMethods.EachUsage(target, handler);
282                    }
283                    else if (_equalityComparer.Equals(target, Constants.Any))
284                    {
285                        return _sourcesTreeMethods.EachUsage(source, handler);
286                    }
287                    else //if(source != Any && target != Any)
288                    {
289                        var link = _sourcesTreeMethods.Search(source, target);
290                        return _equalityComparer.Equals(link, Constants.Null) ?
                            ↪ Constants.Continue : handler(GetLinkStruct(link));
291                    }
292                }
293                else
294                {
295                    if (!Exists(index))
296                    {
297                        return Constants.Continue;
298                    }
299                    if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ _equalityComparer.Equals(target, Constants.Any))
300                    {
301                        return handler(GetLinkStruct(index));
302                    }
303                    ref var storedLinkValue = ref GetLinkUnsafe(index);
304                    if (!_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ !_equalityComparer.Equals(target, Constants.Any))
305                    {
306                        if (_equalityComparer.Equals(storedLinkValue.Source, source) &&
307                            _equalityComparer.Equals(storedLinkValue.Target, target))
308                        {
309                            return handler(GetLinkStruct(index));
310                        }
311                        return Constants.Continue;
312                    }
313                    var value = default(TLink);
314                    if (_equalityComparer.Equals(source, Constants.Any))
315                    {
316                        value = target;
317                    }
318                    if (_equalityComparer.Equals(target, Constants.Any))
319                    {
320                        value = source;
321                    }
322                    if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
323                        _equalityComparer.Equals(storedLinkValue.Target, value))
324                    {
325                        return handler(GetLinkStruct(index));
326                    }
327                    return Constants.Continue;
```

```csharp
                    }
                }
                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> restrictions, IList<TLink> substitution)
        {
            var linkIndex = restrictions[Constants.IndexPart];
            ref var link = ref GetLinkUnsafe(linkIndex);
            ref var firstAsSource = ref AsRef<LinksHeader<TLink>>(_header).FirstAsSource;
            ref var firstAsTarget = ref AsRef<LinksHeader<TLink>>(_header).FirstAsTarget;
            // Будет корректно работать только в том случае, если пространство выделенной связи
            ↪  предварительно заполнено нулями
            if (!_equalityComparer.Equals(link.Source, Constants.Null))
            {
                _sourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
            }
            if (!_equalityComparer.Equals(link.Target, Constants.Null))
            {
                _targetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
            }
            link.Source = substitution[Constants.SourcePart];
            link.Target = substitution[Constants.TargetPart];
            if (!_equalityComparer.Equals(link.Source, Constants.Null))
            {
                _sourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
            }
            if (!_equalityComparer.Equals(link.Target, Constants.Null))
            {
                _targetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
            }
            return linkIndex;
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        internal ref RawLink<TLink> GetLinkUnsafe(TLink linkIndex) => ref
        ↪  AsRef<RawLink<TLink>>(_links + LinkSizeInBytes * (Integer<TLink>)linkIndex);

        /// <remarks>
        /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
        ↪  пространство
        /// </remarks>
        public TLink Create(IList<TLink> restrictions)
        {
            ref var header = ref AsRef<LinksHeader<TLink>>(_header);
            var freeLink = header.FirstFreeLink;
            if (!_equalityComparer.Equals(freeLink, Constants.Null))
            {
                _unusedLinksListMethods.Detach(freeLink);
            }
            else
            {
                var maximumPossibleInnerReference =
                ↪  Constants.PossibleInnerReferencesRange.Maximum;
                if (_comparer.Compare(header.AllocatedLinks, maximumPossibleInnerReference) > 0)
                {
                    throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
                }
                if (_comparer.Compare(header.AllocatedLinks, Decrement(header.ReservedLinks)) >=
                ↪  0)
                {
                    _memory.ReservedCapacity += _memoryReservationStep;
                    SetPointers(_memory);
                    header.ReservedLinks = (Integer<TLink>)(_memory.ReservedCapacity /
                    ↪  LinkSizeInBytes);
```

```csharp
                }
                header.AllocatedLinks = Increment(header.AllocatedLinks);
                _memory.UsedCapacity += LinkSizeInBytes;
                freeLink = header.AllocatedLinks;
            }
            return freeLink;
        }

        public void Delete(IList<TLink> restrictions)
        {
            ref var header = ref AsRef<LinksHeader<TLink>>(_header);
            var link = restrictions[Constants.IndexPart];
            if (_comparer.Compare(link, header.AllocatedLinks) < 0)
            {
                _unusedLinksListMethods.AttachAsFirst(link);
            }
            else if (_equalityComparer.Equals(link, header.AllocatedLinks))
            {
                header.AllocatedLinks = Decrement(header.AllocatedLinks);
                _memory.UsedCapacity -= LinkSizeInBytes;
                // 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 ((_comparer.Compare(header.AllocatedLinks, Integer<TLink>.Zero) > 0) &&
                    IsUnusedLink(header.AllocatedLinks))
                {
                    _unusedLinksListMethods.Detach(header.AllocatedLinks);
                    header.AllocatedLinks = Decrement(header.AllocatedLinks);
                    _memory.UsedCapacity -= LinkSizeInBytes;
                }
            }
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsUnusedLink(TLink link)
            => _equalityComparer.Equals(AsRef<LinksHeader<TLink>>(_header).FirstFreeLink, link)
            || (_equalityComparer.Equals(GetLinkUnsafe(link).SizeAsSource, Constants.Null)
            && !_equalityComparer.Equals(GetLinkUnsafe(link).Source, Constants.Null));

        #region DisposableBase

        protected override bool AllowMultipleDisposeCalls => true;
```

```
472        protected override void Dispose(bool manual, bool wasDisposed)
473        {
474            if (!wasDisposed)
475            {
476                SetPointers(null);
477                _memory.DisposeIfPossible();
478            }
479        }
480
481        #endregion
482    }
483 }
```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksAVLBalancedTreeMethodsBase.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 static System.Runtime.CompilerServices.Unsafe;
 7
 8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 9
10 namespace Platform.Data.Doublets.ResizableDirectMemory
11 {
12     public unsafe abstract class UInt64LinksAVLBalancedTreeMethodsBase :
    ↪ SizedAndThreadedAVLBalancedTreeMethods<ulong>
13     {
14         private readonly UInt64ResizableDirectMemoryLinks _memory;
15         private readonly LinksConstants<ulong> _constants;
16         internal readonly UInt64RawLink* _links;
17         internal readonly UInt64LinksHeader* _header;
18
19         internal UInt64LinksAVLBalancedTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory,
    ↪ UInt64RawLink* links, UInt64LinksHeader* header)
20         {
21             _links = links;
22             _header = header;
23             _memory = memory;
24             _constants = memory.Constants;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override ulong GetZero() => 0UL;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override bool EqualToZero(ulong value) => value == 0UL;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override bool IsEquals(ulong first, ulong second) => first == second;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override bool GreaterThanZero(ulong value) => value > 0UL;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override bool GreaterThan(ulong first, ulong second) => first > second;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
    ↪ always true for ulong
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is
    ↪ always >= 0 for ulong
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
    ↪ for ulong
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected override bool LessThan(ulong first, ulong second) => first < second;
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override ulong Increment(ulong value) => ++value;
```

```csharp
62
63        [MethodImpl(MethodImplOptions.AggressiveInlining)]
64        protected override ulong Decrement(ulong value) => --value;
65
66        [MethodImpl(MethodImplOptions.AggressiveInlining)]
67        protected override ulong Add(ulong first, ulong second) => first + second;
68
69        [MethodImpl(MethodImplOptions.AggressiveInlining)]
70        protected override ulong Subtract(ulong first, ulong second) => first - second;
71
72        [MethodImpl(MethodImplOptions.AggressiveInlining)]
73        protected abstract ulong GetTreeRoot();
74
75        [MethodImpl(MethodImplOptions.AggressiveInlining)]
76        protected abstract ulong GetBasePartValue(ulong link);
77
78        [MethodImpl(MethodImplOptions.AggressiveInlining)]
79        protected abstract bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
          ↪  ulong secondSource, ulong secondTarget);
80
81        [MethodImpl(MethodImplOptions.AggressiveInlining)]
82        protected abstract bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
          ↪  ulong secondSource, ulong secondTarget);
83
84        [MethodImpl(MethodImplOptions.AggressiveInlining)]
85        protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
86        {
87            ref var firstLink = ref _links[first];
88            ref var secondLink = ref _links[second];
89            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
              ↪  secondLink.Source, secondLink.Target);
90        }
91
92        [MethodImpl(MethodImplOptions.AggressiveInlining)]
93        protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
94        {
95            ref var firstLink = ref _links[first];
96            ref var secondLink = ref _links[second];
97            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
              ↪  secondLink.Source, secondLink.Target);
98        }
99
100        [MethodImpl(MethodImplOptions.AggressiveInlining)]
101        protected static ulong GetSizeValue(ulong value) => unchecked((value & 4294967264UL) >>
          ↪  5);
102
103        [MethodImpl(MethodImplOptions.AggressiveInlining)]
104        protected static void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
          ↪  unchecked((storedValue & 31UL) | ((size & 134217727UL) << 5));
105
106        [MethodImpl(MethodImplOptions.AggressiveInlining)]
107        protected static bool GetLeftIsChildValue(ulong value) => unchecked((value & 16UL) >> 4
          ↪  == 1UL);
108
109        [MethodImpl(MethodImplOptions.AggressiveInlining)]
110        protected static void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
          ↪  storedValue = unchecked((storedValue & 4294967279UL) | ((As<bool, byte>(ref value) &
          ↪  1UL) << 4));
111
112        [MethodImpl(MethodImplOptions.AggressiveInlining)]
113        protected static bool GetRightIsChildValue(ulong value) => unchecked((value & 8UL) >> 3
          ↪  == 1UL);
114
115        [MethodImpl(MethodImplOptions.AggressiveInlining)]
116        protected static void SetRightIsChildValue(ref ulong storedValue, bool value) =>
          ↪  storedValue = unchecked((storedValue & 4294967287UL) | ((As<bool, byte>(ref value) &
          ↪  1UL) << 3));
117
118        [MethodImpl(MethodImplOptions.AggressiveInlining)]
119        protected static sbyte GetBalanceValue(ulong value) => unchecked((sbyte)((value & 7UL) |
          ↪  0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
          ↪  sbyte
120
121        [MethodImpl(MethodImplOptions.AggressiveInlining)]
122        protected static void SetBalanceValue(ref ulong storedValue, sbyte value) => storedValue
          ↪  = unchecked((storedValue & 4294967288UL) | ((ulong)(((((byte)value >> 5) & 4) | value
          ↪  & 3) & 7UL));
123
```

```csharp
124          public ulong this[ulong index]
125          {
126              get
127              {
128                  var root = GetTreeRoot();
129                  if (index >= GetSize(root))
130                  {
131                      return 0;
132                  }
133                  while (root != 0)
134                  {
135                      var left = GetLeftOrDefault(root);
136                      var leftSize = GetSizeOrZero(left);
137                      if (index < leftSize)
138                      {
139                          root = left;
140                          continue;
141                      }
142                      if (index == leftSize)
143                      {
144                          return root;
145                      }
146                      root = GetRightOrDefault(root);
147                      index -= leftSize + 1;
148                  }
149                  return 0; // TODO: Impossible situation exception (only if tree structure broken)
150              }
151          }
152
153          /// <summary>
154          /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
     ↪   (концом).
155          /// </summary>
156          /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
157          /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
158          /// <returns>Индекс искомой связи.</returns>
159          public ulong Search(ulong source, ulong target)
160          {
161              var root = GetTreeRoot();
162              while (root != 0)
163              {
164                  var rootLink = _links[root];
165                  var rootSource = rootLink.Source;
166                  var rootTarget = rootLink.Target;
167                  if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
     ↪   node.Key < root.Key
168                  {
169                      root = GetLeftOrDefault(root);
170                  }
171                  else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
     ↪   node.Key > root.Key
172                  {
173                      root = GetRightOrDefault(root);
174                  }
175                  else // node.Key == root.Key
176                  {
177                      return root;
178                  }
179              }
180              return 0;
181          }
182
183          // TODO: Return indices range instead of references count
184          public ulong CountUsages(ulong link)
185          {
186              var root = GetTreeRoot();
187              var total = GetSize(root);
188              var totalRightIgnore = 0UL;
189              while (root != 0)
190              {
191                  var @base = GetBasePartValue(root);
192                  if (@base <= link)
193                  {
194                      root = GetRightOrDefault(root);
195                  }
196                  else
197                  {
198                      totalRightIgnore += GetRightSize(root) + 1;
199                      root = GetLeftOrDefault(root);
```

```
200                    }
201                }
202                root = GetTreeRoot();
203                var totalLeftIgnore = 0UL;
204                while (root != 0)
205                {
206                    var @base = GetBasePartValue(root);
207                    if (@base >= link)
208                    {
209                        root = GetLeftOrDefault(root);
210                    }
211                    else
212                    {
213                        totalLeftIgnore += GetLeftSize(root) + 1;
214                        root = GetRightOrDefault(root);
215                    }
216                }
217                return total - totalRightIgnore - totalLeftIgnore;
218            }
219
220            public ulong EachUsage(ulong link, Func<IList<ulong>, ulong> handler)
221            {
222                var root = GetTreeRoot();
223                if (root == 0)
224                {
225                    return _constants.Continue;
226                }
227                ulong first = 0, current = root;
228                while (current != 0)
229                {
230                    var @base = GetBasePartValue(current);
231                    if (@base >= link)
232                    {
233                        if (@base == link)
234                        {
235                            first = current;
236                        }
237                        current = GetLeftOrDefault(current);
238                    }
239                    else
240                    {
241                        current = GetRightOrDefault(current);
242                    }
243                }
244                if (first != 0)
245                {
246                    current = first;
247                    while (true)
248                    {
249                        if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
250                        {
251                            return _constants.Break;
252                        }
253                        current = GetNext(current);
254                        if (current == 0 || GetBasePartValue(current) != link)
255                        {
256                            break;
257                        }
258                    }
259                }
260                return _constants.Continue;
261            }
262
263            protected override void PrintNodeValue(ulong node, StringBuilder sb)
264            {
265                ref var link = ref _links[node];
266                sb.Append(' ');
267                sb.Append(link.Source);
268                sb.Append('-');
269                sb.Append('>');
270                sb.Append(link.Target);
271            }
272        }
273    }
```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksHeader.cs

```
1    namespace Platform.Data.Doublets.ResizableDirectMemory
2    {
3        internal struct UInt64LinksHeader
```

```
   4        {
   5            public ulong AllocatedLinks;
   6            public ulong ReservedLinks;
   7            public ulong FreeLinks;
   8            public ulong FirstFreeLink;
   9            public ulong FirstAsSource;
  10            public ulong FirstAsTarget;
  11            public ulong LastFreeLink;
  12            public ulong Reserved8;
  13        }
  14    }
```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksSourcesAVLBalancedTreeMethods.cs

```
   1    using System.Runtime.CompilerServices;
   2
   3    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   4
   5    namespace Platform.Data.Doublets.ResizableDirectMemory
   6    {
   7        public unsafe class UInt64LinksSourcesAVLBalancedTreeMethods :
           ↪ UInt64LinksAVLBalancedTreeMethodsBase, ILinksTreeMethods<ulong>
   8        {
   9            internal UInt64LinksSourcesAVLBalancedTreeMethods(UInt64ResizableDirectMemoryLinks
           ↪ memory, UInt64RawLink* links, UInt64LinksHeader* header) : base(memory, links,
           ↪ header) { }
  10
  11            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  12            protected override ref ulong GetLeftReference(ulong node) => ref
           ↪ _links[node].LeftAsSource;
  13
  14            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  15            protected override ref ulong GetRightReference(ulong node) => ref
           ↪ _links[node].RightAsSource;
  16
  17            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  18            protected override ulong GetLeft(ulong node) => _links[node].LeftAsSource;
  19
  20            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  21            protected override ulong GetRight(ulong node) => _links[node].RightAsSource;
  22
  23            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  24            protected override void SetLeft(ulong node, ulong left) => _links[node].LeftAsSource =
           ↪ left;
  25
  26            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  27            protected override void SetRight(ulong node, ulong right) => _links[node].RightAsSource
           ↪ = right;
  28
  29            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  30            protected override ulong GetSize(ulong node) => GetSizeValue(_links[node].SizeAsSource);
  31
  32            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  33            protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
           ↪ _links[node].SizeAsSource, size);
  34
  35            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  36            protected override bool GetLeftIsChild(ulong node) =>
           ↪ GetLeftIsChildValue(_links[node].SizeAsSource);
  37
  38            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  39            protected override void SetLeftIsChild(ulong node, bool value) =>
           ↪ SetLeftIsChildValue(ref _links[node].SizeAsSource, value);
  40
  41            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  42            protected override bool GetRightIsChild(ulong node) =>
           ↪ GetRightIsChildValue(_links[node].SizeAsSource);
  43
  44            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  45            protected override void SetRightIsChild(ulong node, bool value) =>
           ↪ SetRightIsChildValue(ref _links[node].SizeAsSource, value);
  46
  47            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  48            protected override sbyte GetBalance(ulong node) =>
           ↪ GetBalanceValue(_links[node].SizeAsSource);
  49
  50            [MethodImpl(MethodImplOptions.AggressiveInlining)]
  51            protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
           ↪ _links[node].SizeAsSource, value);
  52
```

```
53      [MethodImpl(MethodImplOptions.AggressiveInlining)]
54      protected override ulong GetTreeRoot() => _header->FirstAsSource;
55
56      [MethodImpl(MethodImplOptions.AggressiveInlining)]
57      protected override ulong GetBasePartValue(ulong link) => _links[link].Source;
58
59      [MethodImpl(MethodImplOptions.AggressiveInlining)]
60      protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
61          => firstSource < secondSource || (firstSource == secondSource && firstTarget <
            ↪  secondTarget);
62
63      [MethodImpl(MethodImplOptions.AggressiveInlining)]
64      protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
65          => firstSource > secondSource || (firstSource == secondSource && firstTarget >
            ↪  secondTarget);
66
67      [MethodImpl(MethodImplOptions.AggressiveInlining)]
68      protected override void ClearNode(ulong node)
69      {
70          ref UInt64RawLink link = ref _links[node];
71          link.LeftAsSource = 0UL;
72          link.RightAsSource = 0UL;
73          link.SizeAsSource = 0UL;
74      }
75      }
76  }
```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksTargetsAVLBalancedTreeMethods.cs

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.ResizableDirectMemory
6   {
7       public unsafe class UInt64LinksTargetsAVLBalancedTreeMethods :
        ↪  UInt64LinksAVLBalancedTreeMethodsBase, ILinksTreeMethods<ulong>
8       {
9           internal UInt64LinksTargetsAVLBalancedTreeMethods(UInt64ResizableDirectMemoryLinks
            ↪  memory, UInt64RawLink* links, UInt64LinksHeader* header) : base(memory, links,
            ↪  header) { }
10
11          //protected override IntPtr GetLeft(ulong node) => new IntPtr(&Links[node].LeftAsTarget);
12
13          //protected override IntPtr GetRight(ulong node) => new
            ↪  IntPtr(&Links[node].RightAsTarget);
14
15          //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
16
17          //protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
            ↪  left;
18
19          //protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget
            ↪  = right;
20
21          //protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
            ↪  size;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override ref ulong GetLeftReference(ulong node) => ref
            ↪  _links[node].LeftAsTarget;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override ref ulong GetRightReference(ulong node) => ref
            ↪  _links[node].RightAsTarget;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override ulong GetLeft(ulong node) => _links[node].LeftAsTarget;
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override ulong GetRight(ulong node) => _links[node].RightAsTarget;
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override void SetLeft(ulong node, ulong left) => _links[node].LeftAsTarget =
            ↪  left;
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                protected override ulong GetSize(ulong node) => GetSizeValue(_links[node].SizeAsTarget);

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
                ↪  _links[node].SizeAsTarget, size);

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                protected override bool GetLeftIsChild(ulong node) =>
                ↪  GetLeftIsChildValue(_links[node].SizeAsTarget);

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                protected override void SetLeftIsChild(ulong node, bool value) =>
                ↪  SetLeftIsChildValue(ref _links[node].SizeAsTarget, value);

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                protected override bool GetRightIsChild(ulong node) =>
                ↪  GetRightIsChildValue(_links[node].SizeAsTarget);

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                protected override void SetRightIsChild(ulong node, bool value) =>
                ↪  SetRightIsChildValue(ref _links[node].SizeAsTarget, value);

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                protected override sbyte GetBalance(ulong node) =>
                ↪  GetBalanceValue(_links[node].SizeAsTarget);

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
                ↪  _links[node].SizeAsTarget, value);

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                protected override ulong GetTreeRoot() => _header->FirstAsTarget;

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                protected override ulong GetBasePartValue(ulong link) => _links[link].Target;

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

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

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                protected override void ClearNode(ulong node)
                {
                    ref UInt64RawLink link = ref _links[node];
                    link.LeftAsTarget = 0UL;
                    link.RightAsTarget = 0UL;
                    link.SizeAsTarget = 0UL;
                }
            }
        }
```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64RawLink.cs

```csharp
namespace Platform.Data.Doublets.ResizableDirectMemory
{
    internal struct UInt64RawLink
    {
        public ulong Source;
        public ulong Target;
        public ulong LeftAsSource;
        public ulong RightAsSource;
        public ulong SizeAsSource;
        public ulong LeftAsTarget;
        public ulong RightAsTarget;
        public ulong SizeAsTarget;
    }
}
```

```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;

#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 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(UInt64RawLink);

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

        private readonly long _memoryReservationStep;

        private readonly IResizableDirectMemory _memory;
        private UInt64LinksHeader* _header;
        private UInt64RawLink* _links;

        private ILinksTreeMethods<id> _targetsTreeMethods;
        private ILinksTreeMethods<id> _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 ILinksListMethods<id> _unusedLinksListMethods;

        /// <summary>
        /// Возвращает общее число связей находящихся в хранилище.
        /// </summary>
        private id Total => _header->AllocatedLinks - _header->FreeLinks;

        // TODO: Дать возможность переопределять в конструкторе
        public LinksConstants<id> 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<LinksConstants<id>>.Instance;
            _memory = memory;
            _memoryReservationStep = memoryReservationStep;
            if (memory.ReservedCapacity < memoryReservationStep)
            {
                memory.ReservedCapacity = memoryReservationStep;
```

```csharp
            }
            SetPointers(_memory);
            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
            _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(UInt64RawLink)) +
            ↪    sizeof(UInt64LinksHeader);
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(UInt64LinksHeader))
            ↪    / sizeof(UInt64RawLink));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public id Count(IList<id> restrictions)
        {
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
            if (restrictions.Count == 0)
            {
                return Total;
            }
            if (restrictions.Count == 1)
            {
                var index = restrictions[Constants.IndexPart];
                if (index == Constants.Any)
                {
                    return Total;
                }
                return Exists(index) ? 1UL : 0UL;
            }
            if (restrictions.Count == 2)
            {
                var index = restrictions[Constants.IndexPart];
                var value = restrictions[1];
                if (index == Constants.Any)
                {
                    if (value == Constants.Any)
                    {
                        return Total; // Any - как отсутствие ограничения
                    }
                    return _sourcesTreeMethods.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);
                    }
```

```csharp
                        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;
                        }
                        var storedLinkValue = GetLinkUnsafe(index);
                        if (source != Constants.Any && target != Constants.Any)
                        {
                            if (storedLinkValue->Source == source &&
                                storedLinkValue->Target == target)
                            {
                                return 1;
                            }
                            return 0;
                        }
                        var value = default(id);
                        if (source == Constants.Any)
                        {
                            value = target;
                        }
                        if (target == Constants.Any)
                        {
                            value = source;
                        }
                        if (storedLinkValue->Source == value ||
                            storedLinkValue->Target == value)
                        {
                            return 1;
                        }
                        return 0;
                    }
                }
                throw new NotSupportedException("Другие размеры и способы ограничений не
                    поддерживаются.");
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
            {
                if (restrictions.Count == 0)
                {
                    for (id link = 1; link <= _header->AllocatedLinks; link++)
                    {
                        if (Exists(link))
                        {
                            if (handler(GetLinkStruct(link)) == Constants.Break)
                            {
                                return Constants.Break;
                            }
                        }
                    }
                    return Constants.Continue;
                }
                if (restrictions.Count == 1)
                {
                    var index = restrictions[Constants.IndexPart];
                    if (index == Constants.Any)
                    {
                        return Each(handler, ArrayPool<ulong>.Empty);
                    }
                    if (!Exists(index))
                    {
                        return Constants.Continue;
                    }
                    return handler(GetLinkStruct(index));
                }
                if (restrictions.Count == 2)
```

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

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

        /// <remarks>
        /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
        ↪ в другом месте (но не в менеджере памяти, а в логике Links)
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public id Update(IList<id> restrictions, IList<id> substitution)
        {
            var linkIndex = restrictions[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(ref _header->FirstAsSource, linkIndex);
            }
            if (link->Target != Constants.Null)
            {
                _targetsTreeMethods.Detach(ref _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 = substitution[Constants.SourcePart];
            link->Target = substitution[Constants.TargetPart];
            if (link->Source != Constants.Null)
            {
                _sourcesTreeMethods.Attach(ref _header->FirstAsSource, linkIndex);
            }
            if (link->Target != Constants.Null)
            {
                _targetsTreeMethods.Attach(ref _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)]
        public IList<id> GetLinkStruct(id linkIndex)
        {
            var link = GetLinkUnsafe(linkIndex);
            return new UInt64Link(linkIndex, link->Source, link->Target);
        }
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        internal UInt64RawLink* GetLinkUnsafe(id linkIndex) => &_links[linkIndex];

        /// <remarks>
        /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
        ↪ пространство
        /// </remarks>
        public id Create(IList<id> restritions)
        {
            var freeLink = _header->FirstFreeLink;
            if (freeLink != Constants.Null)
            {
                _unusedLinksListMethods.Detach(freeLink);
            }
            else
            {
                var maximumPossibleInnerReference =
                ↪ Constants.PossibleInnerReferencesRange.Maximum;
                if (_header->AllocatedLinks > maximumPossibleInnerReference)
                {
                    throw new LinksLimitReachedException<id>(maximumPossibleInnerReference);
                }
                if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
                {
                    _memory.ReservedCapacity += _memoryReservationStep;
                    SetPointers(_memory);
                    _header->ReservedLinks = (id)(_memory.ReservedCapacity /
                    ↪ sizeof(UInt64RawLink));
                }
                _header->AllocatedLinks++;
                _memory.UsedCapacity += sizeof(UInt64RawLink);
                freeLink = _header->AllocatedLinks;
            }
            return freeLink;
        }

        public void Delete(IList<id> restrictions)
        {
            var link = restrictions[Constants.IndexPart];
            if (link < _header->AllocatedLinks)
            {
                _unusedLinksListMethods.AttachAsFirst(link);
            }
            else if (link == _header->AllocatedLinks)
            {
                _header->AllocatedLinks--;
                _memory.UsedCapacity -= sizeof(UInt64RawLink);
                // 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(UInt64RawLink);
                }
            }
        }

        /// <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
            {
```

```
452          _header = (UInt64LinksHeader*)(void*)memory.Pointer;
453          _links = (UInt64RawLink*)(void*)memory.Pointer;
454          _sourcesTreeMethods = new UInt64LinksSourcesAVLBalancedTreeMethods(this, _links,
          ↪  _header);
455          _targetsTreeMethods = new UInt64LinksTargetsAVLBalancedTreeMethods(this, _links,
          ↪  _header);
456          _unusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
457        }
458      }
459
460        [MethodImpl(MethodImplOptions.AggressiveInlining)]
461        private bool Exists(id link) => link >= Constants.PossibleInnerReferencesRange.Minimum
          ↪  && link <= _header->AllocatedLinks && !IsUnusedLink(link);
462
463        [MethodImpl(MethodImplOptions.AggressiveInlining)]
464        private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
465                                    || (_links[link].SizeAsSource == Constants.Null &&
                                        ↪  _links[link].Source != Constants.Null);
466
467        #region Disposable
468
469        protected override bool AllowMultipleDisposeCalls => true;
470
471        protected override void Dispose(bool manual, bool wasDisposed)
472        {
473            if (!wasDisposed)
474            {
475                SetPointers(null);
476                _memory.DisposeIfPossible();
477            }
478        }
479
480        #endregion
481    }
482 }
```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64UnusedLinksListMethods.cs

```csharp
1  using Platform.Collections.Methods.Lists;
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.ResizableDirectMemory
7  {
8      public unsafe class UInt64UnusedLinksListMethods : CircularDoublyLinkedListMethods<ulong>,
        ↪  ILinksListMethods<ulong>
9      {
10          private readonly UInt64RawLink* _links;
11          private readonly UInt64LinksHeader* _header;
12
13          internal UInt64UnusedLinksListMethods(UInt64RawLink* links, UInt64LinksHeader* header)
14          {
15              _links = links;
16              _header = header;
17          }
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override ulong GetFirst() => _header->FirstFreeLink;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected override ulong GetLast() => _header->LastFreeLink;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected override ulong GetPrevious(ulong element) => _links[element].Source;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected override ulong GetNext(ulong element) => _links[element].Target;
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected override ulong GetSize() => _header->FreeLinks;
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected override void SetFirst(ulong element) => _header->FirstFreeLink = element;
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected override void SetLast(ulong element) => _header->LastFreeLink = element;
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected override void SetPrevious(ulong element, ulong previous) =>
            ↪  _links[element].Source = previous;
```

```
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          protected override void SetNext(ulong element, ulong next) => _links[element].Target =
    ↪   next;
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          protected override void SetSize(ulong size) => _header->FreeLinks = size;
48      }
49  }
```

./Platform.Data.Doublets/ResizableDirectMemory/UnusedLinksListMethods.cs

```
1   using Platform.Collections.Methods.Lists;
2   using Platform.Numbers;
3   using System.Runtime.CompilerServices;
4   using static System.Runtime.CompilerServices.Unsafe;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.ResizableDirectMemory
9   {
10      public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
    ↪   ILinksListMethods<TLink>
11      {
12          private readonly byte* _links;
13          private readonly byte* _header;
14
15          public UnusedLinksListMethods(byte* links, byte* header)
16          {
17              _links = links;
18              _header = header;
19          }
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          internal virtual ref LinksHeader<TLink> GetHeader() => ref
    ↪   AsRef<LinksHeader<TLink>>(_header);
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          internal virtual ref RawLink<TLink> GetLink(TLink link) => ref
    ↪   AsRef<RawLink<TLink>>((void*)(_links + RawLink<TLink>.SizeInBytes *
    ↪   (Integer<TLink>)link));
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override TLink GetFirst() => GetHeader().FirstFreeLink;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override TLink GetLast() => GetHeader().LastFreeLink;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override TLink GetPrevious(TLink element) => GetLink(element).Source;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override TLink GetNext(TLink element) => GetLink(element).Target;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override TLink GetSize() => GetHeader().FreeLinks;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override void SetFirst(TLink element) => GetHeader().FirstFreeLink = element;
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override void SetLast(TLink element) => GetHeader().LastFreeLink = element;
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override void SetPrevious(TLink element, TLink previous) =>
    ↪   GetLink(element).Source = previous;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void SetNext(TLink element, TLink next) => GetLink(element).Target =
    ↪   next;
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected override void SetSize(TLink size) => GetHeader().FreeLinks = size;
56      }
57  }
```

./Platform.Data.Doublets/Sequences/ArrayExtensions.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.Sequences
 7   {
 8       public static class ArrayExtensions
 9       {
10           public static IList<TLink> ConvertToRestrictionsValues<TLink>(this TLink[] array)
11           {
12               var restrictions = new TLink[array.Length + 1];
13               Array.Copy(array, 0, restrictions, 1, array.Length);
14               return restrictions;
15           }
16       }
17   }
```

./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;
 3   using System.Runtime.CompilerServices;
 4   using Platform.Interfaces;
 5   using Platform.Collections;
 6   using Platform.Singletons;
 7   using Platform.Numbers;
 8   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
```

```csharp
9
10   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12   namespace Platform.Data.Doublets.Sequences.Converters
13   {
14       /// <remarks>
15       /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
         ↪  Links на этапе сжатия.
16       ///     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.
17       ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
         ↪  пар, а так же разом выполнить замену.
18       /// </remarks>
19       public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
20       {
21           private static readonly LinksConstants<TLink> _constants =
             ↪  Default<LinksConstants<TLink>>.Instance;
22           private static readonly EqualityComparer<TLink> _equalityComparer =
             ↪  EqualityComparer<TLink>.Default;
23           private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
24
25           private readonly IConverter<IList<TLink>, TLink> _baseConverter;
26           private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
27           private readonly TLink _minFrequencyToCompress;
28           private readonly bool _doInitialFrequenciesIncrement;
29           private Doublet<TLink> _maxDoublet;
30           private LinkFrequency<TLink> _maxDoubletData;
31
32           private struct HalfDoublet
33           {
34               public TLink Element;
35               public LinkFrequency<TLink> DoubletData;
36
37               public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
38               {
39                   Element = element;
40                   DoubletData = doubletData;
41               }
42
43               public override string ToString() => $"{Element}: ({DoubletData})";
44           }
45
46           public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
             ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
47               : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
48           {
49           }
50
51           public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
             ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
             ↪  doInitialFrequenciesIncrement)
52               : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
                 ↪  doInitialFrequenciesIncrement)
53           {
54           }
55
56           public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
             ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
             ↪  minFrequencyToCompress, bool doInitialFrequenciesIncrement)
57               : base(links)
58           {
59               _baseConverter = baseConverter;
60               _doubletFrequenciesCache = doubletFrequenciesCache;
61               if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
62               {
63                   minFrequencyToCompress = Integer<TLink>.One;
64               }
65               _minFrequencyToCompress = minFrequencyToCompress;
66               _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
67               ResetMaxDoublet();
68           }
69
70           public override TLink Convert(IList<TLink> source) =>
             ↪  _baseConverter.Convert(Compress(source));
71
72           /// <remarks>
73           /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
74           /// Faster version (doublets' frequencies dictionary is not recreated).
75           /// </remarks>
76           private IList<TLink> Compress(IList<TLink> sequence)
```

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

         /// <remarks>
         /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
         /// </remarks>
         private int ReplaceDoublets(HalfDoublet[] copy)
         {
             var oldLength = copy.Length;
             var newLength = copy.Length;
             while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
             {
                 var maxDoubletSource = _maxDoublet.Source;
                 var maxDoubletTarget = _maxDoublet.Target;
                 if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
                 {
                     _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
                 }
                 var maxDoubletReplacementLink = _maxDoubletData.Link;
                 oldLength--;
                 var oldLengthMinusTwo = oldLength - 1;
                 // Substitute all usages
                 int w = 0, r = 0; // (r == read, w == write)
                 for (; r < oldLength; r++)
                 {
                     if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                     ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
                     {
                         if (r > 0)
```

```
153                               {
154                                       var previous = copy[w - 1].Element;
155                                       copy[w - 1].DoubletData.DecrementFrequency();
156                                       copy[w - 1].DoubletData =
                                       ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
                                       ↪ maxDoubletReplacementLink);
157                               }
158                               if (r < oldLengthMinusTwo)
159                               {
160                                       var next = copy[r + 2].Element;
161                                       copy[r + 1].DoubletData.DecrementFrequency();
162                                       copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma┘
                                       ↪ xDoubletReplacementLink,
                                       ↪ next);
163                               }
164                               copy[w++].Element = maxDoubletReplacementLink;
165                               r++;
166                               newLength--;
167                           }
168                           else
169                           {
170                               copy[w++] = copy[r];
171                           }
172                       }
173                       if (w < newLength)
174                       {
175                           copy[w] = copy[r];
176                       }
177                       oldLength = newLength;
178                       ResetMaxDoublet();
179                       UpdateMaxDoublet(copy, newLength);
180                   }
181                   return newLength;
182               }
183
184           [MethodImpl(MethodImplOptions.AggressiveInlining)]
185           private void ResetMaxDoublet()
186           {
187               _maxDoublet = new Doublet<TLink>();
188               _maxDoubletData = new LinkFrequency<TLink>();
189           }
190
191           [MethodImpl(MethodImplOptions.AggressiveInlining)]
192           private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
193           {
194               Doublet<TLink> doublet = default;
195               for (var i = 1; i < length; i++)
196               {
197                   doublet.Source = copy[i - 1].Element;
198                   doublet.Target = copy[i].Element;
199                   UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
200               }
201           }
202
203           [MethodImpl(MethodImplOptions.AggressiveInlining)]
204           private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
205           {
206               var frequency = data.Frequency;
207               var maxFrequency = _maxDoubletData.Frequency;
208               //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
                   ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
                   ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
                   ↪ _maxDoublet.Target)))
209               if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
210                   (_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) */
211               {
212                   _maxDoublet = doublet;
213                   _maxDoubletData = data;
214               }
215           }
216       }
217   }
```

```
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  }
```

```
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

```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 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.Doublets.Unicode;
12
13  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15  namespace Platform.Data.Doublets.Sequences
16  {
17      public class DuplicateSegmentsProvider<TLink> :
            ↪  DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
            ↪  IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
18      {
19          private readonly ILinks<TLink> _links;
20          private readonly ILinks<TLink> _sequences;
21          private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
22          private BitString _visited;
23
```

```csharp
        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, ILinks<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>();
                    var filler = new ListFiller<TLink, TLink>(sequenceElements,
                    ↪  _sequences.Constants.Break);
                    _sequences.Each(filler.AddAllValuesAndReturnConstant, new
                    ↪  LinkAddress<TLink>(linkIndex));
                    if (sequenceElements.Count > 2)
                    {
                        WalkAll(sequenceElements);
                    }
                }
                return _links.Constants.Continue;
            });
            var resultList = _groups.ToList();
            var comparer = Default<ItemComparer>.Instance;
            resultList.Sort(comparer);
#if DEBUG
            foreach (var item in resultList)
            {
                PrintDuplicates(item);
            }
#endif
            return resultList;
        }

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

```csharp
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             var restrictions = segment.ConvertToRestrictionsValues();
105             restrictions[0] = _sequences.Constants.Any;
106             _sequences.Each(sequence =>
107             {
108                 var sequenceIndex = sequence[_sequences.Constants.IndexPart];
109                 duplicates.Add(sequenceIndex);
110                 readAsElement.Add(sequenceIndex);
111                 return _sequences.Constants.Continue;
112             }, restrictions);
113             if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
114             {
115                 return new List<TLink>();
116             }
117             foreach (var duplicate in duplicates)
118             {
119                 var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
120                 _visited.Set(duplicateBitIndex);
121             }
122             if (_sequences is Sequences sequencesExperiments)
123             {
124                 var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
                    ↪ ashSet<ulong>)(object)readAsElement,
                    ↪ (IList<ulong>)segment);
125                 foreach (var partiallyMatchedSequence in partiallyMatched)
126                 {
127                     TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
128                     duplicates.Add(sequenceIndex);
129                 }
130             }
131             duplicates.Sort();
132             return duplicates;
133         }
134
135         private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
136         {
137             if (!(_links is ILinks<ulong> ulongLinks))
138             {
139                 return;
140             }
141             var duplicatesKey = duplicatesItem.Key;
142             var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
143             Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
144             var duplicatesList = duplicatesItem.Value;
145             for (int i = 0; i < duplicatesList.Count; i++)
146             {
147                 ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
148                 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));
149                 Console.WriteLine(formatedSequenceStructure);
150                 var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
                    ↪ ulongLinks);
151                 Console.WriteLine(sequenceString);
152             }
153             Console.WriteLine();
154         }
155     }
156 }
```

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

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

```csharp
using System.Runtime.CompilerServices;
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();
```

```
79                     }
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

```csharp
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)]
```

```csharp
        public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);

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

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

```csharp
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
        IConverter<Doublet<TLink>, TLink>
    {
        private readonly LinkFrequenciesCache<TLink> _cache;
        public
            FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
            cache) => _cache = cache;
        public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
    }
}
```

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

```csharp
using Platform.Interfaces;

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

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

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

        public override TLink Count()
        {
            if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
            {
                return default;
            }
            return base.Count();
        }
    }
}
```

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

```csharp
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Numbers;
using Platform.Data.Sequences;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        protected readonly ILinks<TLink> _links;
        protected readonly TLink _sequenceLink;
        protected readonly TLink _symbol;
        protected TLink _total;

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

```
26          }
27
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

```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.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

```csharp
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/IListExtensions.cs

```
1  using Platform.Collections;
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
7  {
8      public static class IListExtensions
9      {
10         public static TLink[] ExtractValues<TLink>(this IList<TLink> restrictions)
11         {
12             if(restrictions.IsNullOrEmpty() || restrictions.Count == 1)
13             {
14                 return new TLink[0];
15             }
16             var values = new TLink[restrictions.Count - 1];
17             for (int i = 1, j = 0; i < restrictions.Count; i++, j++)
18             {
19                 values[j] = restrictions[i];
20             }
21             return values;
22         }
23
24         public static IList<TLink> ConvertToRestrictionsValues<TLink>(this IList<TLink> list)
25         {
26             var restrictions = new TLink[list.Count + 1];
27             for (int i = 0, j = 1; i < list.Count; i++, j++)
28             {
29                 restrictions[j] = list[i];
30             }
31             return restrictions;
32         }
33     }
34 }
```

./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/ListFiller.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.Sequences
7    {
8        public class ListFiller<TElement, TReturnConstant>
9        {
10           protected readonly List<TElement> _list;
11           protected readonly TReturnConstant _returnConstant;
12
13           public ListFiller(List<TElement> list, TReturnConstant returnConstant)
14           {
15               _list = list;
16               _returnConstant = returnConstant;
17           }
18
```

```csharp
        public ListFiller(List<TElement> list) : this(list, default) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TElement element) => _list.Add(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddAndReturnTrue(TElement element)
        {
            _list.Add(element);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddFirstAndReturnTrue(IList<TElement> collection)
        {
            _list.Add(collection[0]);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAndReturnConstant(TElement element)
        {
            _list.Add(element);
            return _returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddFirstAndReturnConstant(IList<TElement> collection)
        {
            _list.Add(collection[0]);
            return _returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAllValuesAndReturnConstant(IList<TElement> collection)
        {
            for (int i = 1; i < collection.Count; i++)
            {
                _list.Add(collection[i]);
            }
            return _returnConstant;
        }
    }
}
```

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

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using Platform.Collections;
using Platform.Collections.Lists;
using Platform.Threading.Synchronization;
using Platform.Singletons;
using LinkIndex = System.UInt64;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Collections.Stacks;
using Platform.Collections.Arrays;

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// Представляет коллекцию последовательностей связей.
    /// </summary>
    /// <remarks>
    /// Обязательно реализовать атомарность каждого публичного метода.
    ///
    /// TODO:
    ///
    /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
    /// через естественную группировку по unicode типам, все whitespace вместе, все символы
    ///   вместе, все числа вместе и т.п.
    /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
    ///   графа)
    ///
    /// x*y - найти все связи между, в последовательностях любой формы, если не стоит
    ///   ограничитель на то, что является последовательностью, а что нет,
```

```
31    /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
   ↪  порядке.
32    ///
33    /// Рост последовательности слева и справа.
34    /// Поиск со звёздочкой.
35    /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
36    /// так же проблема может быть решена при реализации дистанционных триггеров.
37    /// Нужны ли уникальные указатели вообще?
38    /// Что если обращение к информации будет происходить через содержимое всегда?
39    ///
40    /// Писать тесты.
41    ///
42    ///
43    /// Можно убрать зависимость от конкретной реализации Links,
44    /// на зависимость от абстрактного элемента, который может быть представлен несколькими
   ↪  способами.
45    ///
46    /// Можно ли как-то сделать один общий интерфейс
47    ///
48    ///
49    /// Блокчейн и/или гит для распределённой записи транзакций.
50    ///
51    /// </remarks>
52    public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
   ↪  (после завершения реализации Sequences)
53    {
54        /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
   ↪  связей.</summary>
55        public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
56
57        public SequencesOptions<LinkIndex> Options { get; }
58        public SynchronizedLinks<LinkIndex> Links { get; }
59        private readonly ISynchronization _sync;
60
61        public LinksConstants<LinkIndex> Constants { get; }
62
63        public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
64        {
65            Links = links;
66            _sync = links.SyncRoot;
67            Options = options;
68            Options.ValidateOptions();
69            Options.InitOptions(Links);
70            Constants = Default<LinksConstants<LinkIndex>>.Instance;
71        }
72
73        public Sequences(SynchronizedLinks<LinkIndex> links)
74            : this(links, new SequencesOptions<LinkIndex>())
75        {
76        }
77
78        public bool IsSequence(LinkIndex sequence)
79        {
80            return _sync.ExecuteReadOperation(() =>
81            {
82                if (Options.UseSequenceMarker)
83                {
84                    return Options.MarkedSequenceMatcher.IsMatched(sequence);
85                }
86                return !Links.Unsync.IsPartialPoint(sequence);
87            });
88        }
89
90        [MethodImpl(MethodImplOptions.AggressiveInlining)]
91        private LinkIndex GetSequenceByElements(LinkIndex sequence)
92        {
93            if (Options.UseSequenceMarker)
94            {
95                return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
96            }
97            return sequence;
98        }
99
100        private LinkIndex GetSequenceElements(LinkIndex sequence)
101        {
102            if (Options.UseSequenceMarker)
103            {
104                var linkContents = new UInt64Link(Links.GetLink(sequence));
105                if (linkContents.Source == Options.SequenceMarkerLink)
```

```csharp
                {
                    return linkContents.Target;
                }
                if (linkContents.Target == Options.SequenceMarkerLink)
                {
                    return linkContents.Source;
                }
            }
            return sequence;
        }

        #region Count

        public LinkIndex Count(IList<LinkIndex> restrictions)
        {
            if (restrictions.IsNullOrEmpty())
            {
                return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
            }
            if (restrictions.Count == 1) // Первая связь это адрес
            {
                var sequenceIndex = restrictions[0];
                if (sequenceIndex == Constants.Null)
                {
                    return 0;
                }
                if (sequenceIndex == Constants.Any)
                {
                    return Count(null);
                }
                if (Options.UseSequenceMarker)
                {
                    return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
                }
                return Links.Exists(sequenceIndex) ? 1UL : 0;
            }
            throw new NotImplementedException();
        }

        private LinkIndex CountUsages(params LinkIndex[] 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 LinkIndex Create(IList<LinkIndex> restrictions)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (restrictions.IsNullOrEmpty())
                {
                    return Constants.Null;
                }
                Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
```

```csharp
185                     return CreateCore(restrictions);
186                 });
187         }
188
189         private LinkIndex CreateCore(IList<LinkIndex> restrictions)
190         {
191             LinkIndex[] sequence = restrictions.ExtractValues();
192             if (Options.UseIndex)
193             {
194                 Options.Index.Add(sequence);
195             }
196             var sequenceRoot = default(LinkIndex);
197             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
198             {
199                 var matches = Each(restrictions);
200                 if (matches.Count > 0)
201                 {
202                     sequenceRoot = matches[0];
203                 }
204             }
205             else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
206             {
207                 return CompactCore(sequence);
208             }
209             if (sequenceRoot == default)
210             {
211                 sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
212             }
213             if (Options.UseSequenceMarker)
214             {
215                 Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
216             }
217             return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
218         }
219
220         #endregion
221
222         #region Each
223
224         public List<LinkIndex> Each(IList<LinkIndex> sequence)
225         {
226             var results = new List<LinkIndex>();
227             var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
228             Each(filler.AddFirstAndReturnConstant, sequence);
229             return results;
230         }
231
232         public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↪  restrictions)
233         {
234             return _sync.ExecuteReadOperation(() =>
235             {
236                 if (restrictions.IsNullOrEmpty())
237                 {
238                     return Constants.Continue;
239                 }
240                 Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
241                 if (restrictions.Count == 1)
242                 {
243                     var link = restrictions[0];
244                     var any = Constants.Any;
245                     if (link == any)
246                     {
247                         if (Options.UseSequenceMarker)
248                         {
249                             return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
    ↪  Options.SequenceMarkerLink, any));
250                         }
251                         else
252                         {
253                             return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
    ↪  any));
254                         }
255                     }
256                     var sequence =
    ↪  Options.Walker.Walk(link).ToArray().ConvertToRestrictionsValues();
257                     sequence[0] = link;
258                     return handler(sequence);
```

```csharp
                    }
                    else if (restrictions.Count == 2)
                    {
                        throw new NotImplementedException();
                    }
                    else if (restrictions.Count == 3)
                    {
                        return Links.Unsync.Each(handler, restrictions);
                    }
                    else
                    {
                        var sequence = restrictions.ExtractValues();
                        if (Options.UseIndex && !Options.Index.MightContain(sequence))
                        {
                            return Constants.Break;
                        }
                        return EachCore(handler, sequence);
                    }
                });
        }

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

        private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
        ↪   left, LinkIndex right)
        {
            return Links.Unsync.Each(doublet =>
            {
                var doubletIndex = doublet[Constants.IndexPart];
                if (StepRight(handler, doubletIndex, right) != Constants.Continue)
                {
                    return Constants.Break;
                }
                if (left != doubletIndex)
                {
                    return PartialStepRight(handler, doubletIndex, right);
                }
                return Constants.Continue;
            }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
        }

        private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
        ↪   LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
        ↪   rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
        ↪   Constants.Any));
```

```csharp
327        private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪  right, LinkIndex stepFrom)
328        {
329            var upStep = stepFrom;
330            var firstSource = Links.Unsync.GetTarget(upStep);
331            while (firstSource != right && firstSource != upStep)
332            {
333                upStep = firstSource;
334                firstSource = Links.Unsync.GetSource(upStep);
335            }
336            if (firstSource == right)
337            {
338                return handler(new LinkAddress<LinkIndex>(stepFrom));
339            }
340            return Constants.Continue;
341        }
342
343        private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↪  LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
    ↪  leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
    ↪  right));
344
345        private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪  left, LinkIndex stepFrom)
346        {
347            var upStep = stepFrom;
348            var firstTarget = Links.Unsync.GetSource(upStep);
349            while (firstTarget != left && firstTarget != upStep)
350            {
351                upStep = firstTarget;
352                firstTarget = Links.Unsync.GetTarget(upStep);
353            }
354            if (firstTarget == left)
355            {
356                return handler(new LinkAddress<LinkIndex>(stepFrom));
357            }
358            return Constants.Continue;
359        }
360
361        #endregion
362
363        #region Update
364
365        public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
366        {
367            var sequence = restrictions.ExtractValues();
368            var newSequence = substitution.ExtractValues();
369
370            if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
371            {
372                return Constants.Null;
373            }
374            if (sequence.IsNullOrEmpty())
375            {
376                return Create(substitution);
377            }
378            if (newSequence.IsNullOrEmpty())
379            {
380                Delete(restrictions);
381                return Constants.Null;
382            }
383            return _sync.ExecuteWriteOperation(() =>
384            {
385                Links.EnsureEachLinkIsAnyOrExists(sequence);
386                Links.EnsureEachLinkExists(newSequence);
387                return UpdateCore(sequence, newSequence);
388            });
389        }
390
391        private LinkIndex UpdateCore(LinkIndex[] sequence, LinkIndex[] newSequence)
392        {
393            LinkIndex bestVariant;
394            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
    ↪  !sequence.EqualTo(newSequence))
395            {
396                bestVariant = CompactCore(newSequence);
397            }
398            else
399            {
```

```
400                      bestVariant = CreateCore(newSequence);
401                  }
402                  // TODO: Check all options only ones before loop execution
403                  // Возможно нужно две версии Each, возвращающий фактические последовательности и с
      ↪   маркером,
404                  // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
      ↪   можно получить имея только фактические последовательности.
405                  foreach (var variant in Each(sequence))
406                  {
407                      if (variant != bestVariant)
408                      {
409                          UpdateOneCore(variant, bestVariant);
410                      }
411                  }
412                  return bestVariant;
413              }
414
415          private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
416          {
417              if (Options.UseGarbageCollection)
418              {
419                  var sequenceElements = GetSequenceElements(sequence);
420                  var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
421                  var sequenceLink = GetSequenceByElements(sequenceElements);
422                  var newSequenceElements = GetSequenceElements(newSequence);
423                  var newSequenceLink = GetSequenceByElements(newSequenceElements);
424                  if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
425                  {
426                      if (sequenceLink != Constants.Null)
427                      {
428                          Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
429                      }
430                      Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
431                  }
432                  ClearGarbage(sequenceElementsContents.Source);
433                  ClearGarbage(sequenceElementsContents.Target);
434              }
435              else
436              {
437                  if (Options.UseSequenceMarker)
438                  {
439                      var sequenceElements = GetSequenceElements(sequence);
440                      var sequenceLink = GetSequenceByElements(sequenceElements);
441                      var newSequenceElements = GetSequenceElements(newSequence);
442                      var newSequenceLink = GetSequenceByElements(newSequenceElements);
443                      if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
444                      {
445                          if (sequenceLink != Constants.Null)
446                          {
447                              Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
448                          }
449                          Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
450                      }
451                  }
452                  else
453                  {
454                      if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
455                      {
456                          Links.Unsync.MergeUsages(sequence, newSequence);
457                      }
458                  }
459              }
460          }
461
462          #endregion
463
464          #region Delete
465
466          public void Delete(IList<LinkIndex> restrictions)
467          {
468              _sync.ExecuteWriteOperation(() =>
469              {
470                  var sequence = restrictions.ExtractValues();
471                  // TODO: Check all options only ones before loop execution
472                  foreach (var linkToDelete in Each(sequence))
473                  {
474                      DeleteOneCore(linkToDelete);
475                  }
```

```
476                  });
477              }
478
479          private void DeleteOneCore(LinkIndex link)
480          {
481              if (Options.UseGarbageCollection)
482              {
483                  var sequenceElements = GetSequenceElements(link);
484                  var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
485                  var sequenceLink = GetSequenceByElements(sequenceElements);
486                  if (Options.UseCascadeDelete || CountUsages(link) == 0)
487                  {
488                      if (sequenceLink != Constants.Null)
489                      {
490                          Links.Unsync.Delete(sequenceLink);
491                      }
492                      Links.Unsync.Delete(link);
493                  }
494                  ClearGarbage(sequenceElementsContents.Source);
495                  ClearGarbage(sequenceElementsContents.Target);
496              }
497              else
498              {
499                  if (Options.UseSequenceMarker)
500                  {
501                      var sequenceElements = GetSequenceElements(link);
502                      var sequenceLink = GetSequenceByElements(sequenceElements);
503                      if (Options.UseCascadeDelete || CountUsages(link) == 0)
504                      {
505                          if (sequenceLink != Constants.Null)
506                          {
507                              Links.Unsync.Delete(sequenceLink);
508                          }
509                          Links.Unsync.Delete(link);
510                      }
511                  }
512                  else
513                  {
514                      if (Options.UseCascadeDelete || CountUsages(link) == 0)
515                      {
516                          Links.Unsync.Delete(link);
517                      }
518                  }
519              }
520          }
521
522          #endregion
523
524          #region Compactification
525
526          /// <remarks>
527          /// bestVariant можно выбирать по максимальному числу использований,
528          /// но балансированный позволяет гарантировать уникальность (если есть возможность,
529          /// гарантировать его использование в других местах).
530          /// 
531          /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
532          /// </remarks>
533          public LinkIndex Compact(params LinkIndex[] sequence)
534          {
535              return _sync.ExecuteWriteOperation(() =>
536              {
537                  if (sequence.IsNullOrEmpty())
538                  {
539                      return Constants.Null;
540                  }
541                  Links.EnsureEachLinkExists(sequence);
542                  return CompactCore(sequence);
543              });
544          }
545
546          [MethodImpl(MethodImplOptions.AggressiveInlining)]
547          private LinkIndex CompactCore(params LinkIndex[] sequence) => UpdateCore(sequence,
   ↪  sequence);
548
549          #endregion
550
551          #region Garbage Collection
552
553          /// <remarks>
```

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

        private void ClearGarbage(LinkIndex 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<LinkIndex, bool> handler, LinkIndex 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<LinkIndex>
        {
            private readonly Sequences _sequences;
            private readonly IList<LinkIndex> _patternSequence;
            private readonly HashSet<LinkIndex> _linksInSequence;
            private readonly HashSet<LinkIndex> _results;
            private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
            private readonly HashSet<LinkIndex> _readAsElements;
            private int _filterPosition;

            public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
            ↪ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
            ↪ HashSet<LinkIndex> readAsElements = null)
                : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
            {
                _sequences = sequences;
                _patternSequence = patternSequence;
                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                ↪ Links.Constants.Any && x != ZeroOrMany));
                _results = results;
                _stopableHandler = stopableHandler;
                _readAsElements = readAsElements;
            }

            protected override bool IsElement(LinkIndex 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)
```

```csharp
                    {
                        if (_filterPosition == _patternSequence.Count)
                        {
                            _filterPosition = -2; // Длиннее чем нужно
                            return false;
                        }
                        if (_patternSequence[_filterPosition] != Links.Constants.Any
                         && element != _patternSequence[_filterPosition])
                        {
                            _filterPosition = -1;
                            return false; // Начинается/Продолжается иначе
                        }
                        _filterPosition++;
                        return true;
                    }

                    public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
                    {
                        var sequenceToMatch = restrictions[Links.Constants.IndexPart];
                        if (FullMatch(sequenceToMatch))
                        {
                            _results.Add(sequenceToMatch);
                        }
                    }

                    public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
                    {
                        var sequenceToMatch = restrictions[Links.Constants.IndexPart];
                        if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
                        {
                            return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
                        }
                        return Links.Constants.Continue;
                    }

                    public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
                    {
                        var sequenceToMatch = restrictions[Links.Constants.IndexPart];
                        var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
                        if (sequence != Links.Constants.Null && FullMatch(sequenceToMatch) &&
                        ↪  _results.Add(sequenceToMatch))
                        {
                            return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
                        }
                        return Links.Constants.Continue;
                    }

                    /// <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;
                    }

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

```
705                    }
706                    if (_filterPosition < 0)
707                    {
708                        if (element == _patternSequence[0])
709                        {
710                            _filterPosition = 0;
711                        }
712                    }
713                    return true; // Ищем дальше
714                }
715
716            public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
717            {
718                if (PartialMatch(sequenceToMatch))
719                {
720                    _results.Add(sequenceToMatch);
721                }
722            }
723
724            public LinkIndex HandlePartialMatched(IList<LinkIndex> restrictions)
725            {
726                var sequenceToMatch = restrictions[Links.Constants.IndexPart];
727                if (PartialMatch(sequenceToMatch))
728                {
729                    return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
730                }
731                return Links.Constants.Continue;
732            }
733
734            public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
735            {
736                foreach (var sequenceToMatch in sequencesToMatch)
737                {
738                    if (PartialMatch(sequenceToMatch))
739                    {
740                        _results.Add(sequenceToMatch);
741                    }
742                }
743            }
744
745            public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
     ↪   sequencesToMatch)
746            {
747                foreach (var sequenceToMatch in sequencesToMatch)
748                {
749                    if (PartialMatch(sequenceToMatch))
750                    {
751                        _readAsElements.Add(sequenceToMatch);
752                        _results.Add(sequenceToMatch);
753                    }
754                }
755            }
756        }
757
758        #endregion
759    }
760 }
```

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

```
1  using System;
2  using LinkIndex = System.UInt64;
3  using System.Collections.Generic;
4  using Stack = System.Collections.Generic.Stack<ulong>;
5  using System.Linq;
6  using System.Text;
7  using Platform.Collections;
8  using Platform.Data.Exceptions;
9  using Platform.Data.Sequences;
10 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
11 using Platform.Data.Doublets.Sequences.Walkers;
12 using Platform.Collections.Stacks;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     partial class Sequences
19     {
20         #region Create All Variants (Not Practical)
21
```

```csharp
            /// <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)
            {
                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);
```

```csharp
                });
        }

        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
            {
                var innerSequenceLength = sequence.Length - 1;
                for (var li = 0; li < innerSequenceLength; li++)
                {
                    var left = sequence[li];
                    var right = sequence[li + 1];
                    if (left == 0 && right == 0)
                    {
                        continue;
                    }
                    var linkIndex = li;
                    ulong[] innerSequence = null;
                    Links.Unsync.Each(doublet =>
                    {
                        if (innerSequence == null)
                        {
```

```csharp
                        innerSequence = new ulong[innerSequenceLength];
                        for (var isi = 0; isi < linkIndex; isi++)
                        {
                            innerSequence[isi] = sequence[isi];
                        }
                        for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
                        {
                            innerSequence[isi] = sequence[isi + 1];
                        }
                    }
                    innerSequence[linkIndex] = doublet[Constants.IndexPart];
                    Each1(handler, innerSequence);
                    return Constants.Continue;
                }, Constants.Any, left, right);
            }
        }
    }

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

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

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

```
                            handler(new LinkAddress<LinkIndex>(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(new LinkAddress<LinkIndex>(match));
                        }
                        return true;
                    });
                    //           ._x o_.
                    //           |___|
                    PartialStepRight(x => handler(x), sequence[0], sequence[1]);
                }
                else
                {
                    throw new NotImplementedException();
                }
        }

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

        private void TryStepRightUp(Action<IList<LinkIndex>> 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(new LinkAddress<LinkIndex>(stepFrom));
            }
        }

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

        private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
        {
```

```
328            Links.Unsync.Each(Constants.Any, right, leftStep =>
329            {
330                TryStepLeftUp(handler, left, leftStep);
331                return true;
332            });
333        }
334
335        private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
336        {
337            var upStep = stepFrom;
338            var firstTarget = Links.Unsync.GetSource(upStep);
339            while (firstTarget != left && firstTarget != upStep)
340            {
341                upStep = firstTarget;
342                firstTarget = Links.Unsync.GetTarget(upStep);
343            }
344            if (firstTarget == left)
345            {
346                handler(new LinkAddress<LinkIndex>(stepFrom));
347            }
348        }
349
350        private bool StartsWith(ulong sequence, ulong link)
351        {
352            var upStep = sequence;
353            var firstSource = Links.Unsync.GetSource(upStep);
354            while (firstSource != link && firstSource != upStep)
355            {
356                upStep = firstSource;
357                firstSource = Links.Unsync.GetSource(upStep);
358            }
359            return firstSource == link;
360        }
361
362        private bool EndsWith(ulong sequence, ulong link)
363        {
364            var upStep = sequence;
365            var lastTarget = Links.Unsync.GetTarget(upStep);
366            while (lastTarget != link && lastTarget != upStep)
367            {
368                upStep = lastTarget;
369                lastTarget = Links.Unsync.GetTarget(upStep);
370            }
371            return lastTarget == link;
372        }
373
374        public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
375        {
376            return _sync.ExecuteReadOperation(() =>
377            {
378                var results = new List<ulong>();
379                if (sequence.Length > 0)
380                {
381                    Links.EnsureEachLinkExists(sequence);
382                    var firstElement = sequence[0];
383                    if (sequence.Length == 1)
384                    {
385                        results.Add(firstElement);
386                        return results;
387                    }
388                    if (sequence.Length == 2)
389                    {
390                        var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
391                        if (doublet != Constants.Null)
392                        {
393                            results.Add(doublet);
394                        }
395                        return results;
396                    }
397                    var linksInSequence = new HashSet<ulong>(sequence);
398                    void handler(IList<LinkIndex> result)
399                    {
400                        var resultIndex = result[Links.Constants.IndexPart];
401                        var filterPosition = 0;
402                        StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
                        ↪  Links.Unsync.GetTarget,
403                            x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                        ↪    x =>
404                            {
```

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

                            return true;
                        });
                    if (filterPosition == sequence.Length)
                    {
                        results.Add(resultIndex);
                    }
                }
                if (sequence.Length >= 2)
                {
                    StepRight(handler, sequence[0], sequence[1]);
                }
                var last = sequence.Length - 2;
                for (var i = 1; i < last; i++)
                {
                    PartialStepRight(handler, sequence[i], sequence[i + 1]);
                }
                if (sequence.Length >= 3)
                {
                    StepLeft(handler, sequence[sequence.Length - 2],
                    ↪  sequence[sequence.Length - 1]);
                }
            }
            return results;
        });
    }

    public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            var results = new HashSet<ulong>();
            if (sequence.Length > 0)
            {
                Links.EnsureEachLinkExists(sequence);
                var firstElement = sequence[0];
                if (sequence.Length == 1)
                {
                    results.Add(firstElement);
                    return results;
                }
                if (sequence.Length == 2)
                {
                    var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
                    if (doublet != Constants.Null)
                    {
                        results.Add(doublet);
                    }
                    return results;
                }
                var matcher = new Matcher(this, sequence, results, null);
                if (sequence.Length >= 2)
                {
                    StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
                }
                var last = sequence.Length - 2;
                for (var i = 1; i < last; i++)
                {
                    PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
                    ↪  sequence[i + 1]);
                }
                if (sequence.Length >= 3)
                {
                    StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
                    ↪  sequence[sequence.Length - 1]);
                }
            }
        }
```

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

```csharp
                                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 List<ulong> GetAllPartiallyMatchingSequences0(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 List<ulong>();
                var linksInSequence = new HashSet<ulong>(sequence);
                foreach (var result in results)
                {
                    var filterPosition = -1;
                    StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
                    ↪  Links.Unsync.GetTarget,
                        x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                    ↪  x =>
                        {
                            if (filterPosition == (sequence.Length - 1))
                            {
                                return false;
                            }
                            if (filterPosition >= 0)
                            {
                                if (x == sequence[filterPosition + 1])
                                {
                                    filterPosition++;
                                }
                                else
                                {
                                    return false;
                                }
                            }
                            if (filterPosition < 0)
                            {
                                if (x == sequence[0])
                                {
                                    filterPosition = 0;
                                }
                            }
                            return true;
                        });
                    if (filterPosition == (sequence.Length - 1))
                    {
                        filteredResults.Add(result);
                    }
                }
                return filteredResults;
            }
            return new List<ulong>();
        });
```

```csharp
        }

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

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

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

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

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

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

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

        //            firstResults.IntersectWith(lastResults);

        //            //for (var i = 0; i < sequence.Length; i++)
        //            //    AllUsagesCore(sequence[i], results);

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

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

```csharp
        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 != Constants.Any);
                    var last = sequence.Last(x => x != Constants.Any);
                    AllUsagesCore(first, firstResults);
                    AllUsagesCore(last, lastResults);
                    firstResults.IntersectWith(lastResults);
                    //for (var i = 0; i < sequence.Length; i++)
                    //    AllUsagesCore(sequence[i], results);
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new Matcher(this, sequence, filteredResults, null);
                    matcher.AddAllPartialMatchedToResults(firstResults);
                    return filteredResults;
                }
                return new HashSet<ulong>();
            });
        }

        public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
        ↪  IList<ulong> sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Count > 0)
                {
                    Links.EnsureEachLinkExists(sequence);
                    var results = new HashSet<LinkIndex>();
                    //var nextResults = new HashSet<ulong>();
                    //for (var i = 0; i < sequence.Length; i++)
                    //{
                    //    AllUsagesCore(sequence[i], nextResults);
                    //    if (results.IsNullOrEmpty())
                    //    {
                    //        results = nextResults;
                    //        nextResults = new HashSet<ulong>();
                    //    }
                    //    else
                    //    {
                    //        results.IntersectWith(nextResults);
                    //        nextResults.Clear();
                    //    }
                    //}
                    var collector1 = new AllUsagesCollector1(Links.Unsync, results);
                    collector1.Collect(Links.Unsync.GetLink(sequence[0]));
                    var next = new HashSet<ulong>();
                    for (var i = 1; i < sequence.Count; i++)
                    {
                        var collector = new AllUsagesCollector1(Links.Unsync, next);
                        collector.Collect(Links.Unsync.GetLink(sequence[i]));

                        results.IntersectWith(next);
                        next.Clear();
                    }
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new Matcher(this, sequence, filteredResults, null,
                        ↪  readAsElements);
                    matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
                        ↪  x)); // OrderBy is a Hack
                    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>();
```

```
771    //     var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
       ↪  true; }, readAsElements);
772    //     var last = sequence.Length - 1;
773    //     for (var i = 0; i < last; i++)
774    //     {
775    //         PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
776    //     }
777    //     return results;
778    //}
779
780    public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
781    {
782        return _sync.ExecuteReadOperation(() =>
783        {
784            if (sequence.Length > 0)
785            {
786                Links.EnsureEachLinkExists(sequence);
787                //var firstElement = sequence[0];
788                //if (sequence.Length == 1)
789                //{
790                //    //results.Add(firstElement);
791                //    return results;
792                //}
793                //if (sequence.Length == 2)
794                //{
795                //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
796                //    //if (doublet != Doublets.Links.Null)
797                //    //    results.Add(doublet);
798                //    return results;
799                //}
800                //var lastElement = sequence[sequence.Length - 1];
801                //Func<ulong, bool> handler = x =>
802                //{
803                //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
       ↪  results.Add(x);
804                //    return true;
805                //};
806                //if (sequence.Length >= 2)
807                //    StepRight(handler, sequence[0], sequence[1]);
808                //var last = sequence.Length - 2;
809                //for (var i = 1; i < last; i++)
810                //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
811                //if (sequence.Length >= 3)
812                //    StepLeft(handler, sequence[sequence.Length - 2],
       ↪  sequence[sequence.Length - 1]);
813                //////if (sequence.Length == 1)
814                //////{
815                //////    throw new NotImplementedException(); // all sequences, containing
       ↪  this element?
816                //////}
817                //////if (sequence.Length == 2)
818                //////{
819                //////    var results = new List<ulong>();
820                //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
821                //////    return results;
822                //////}
823                //////var matches = new List<List<ulong>>();
824                //////var last = sequence.Length - 1;
825                //////for (var i = 0; i < last; i++)
826                //////{
827                //////    var results = new List<ulong>();
828                //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
829                //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
830                //////    if (results.Count > 0)
831                //////        matches.Add(results);
832                //////    else
833                //////        return results;
834                //////    if (matches.Count == 2)
835                //////    {
836                //////        var merged = new List<ulong>();
837                //////        for (var j = 0; j < matches[0].Count; j++)
838                //////            for (var k = 0; k < matches[1].Count; k++)
839                //////                CloseInnerConnections(merged.Add, matches[0][j],
       ↪  matches[1][k]);
840                //////        if (merged.Count > 0)
841                //////            matches = new List<List<ulong>> { merged };
842                //////        else
```

```csharp
843                  //////           return new List<ulong>();
844                  //////     }
845                  //////}
846                  //////if (matches.Count > 0)
847                  //////{
848                  //////     var usages = new HashSet<ulong>();
849                  //////     for (int i = 0; i < sequence.Length; i++)
850                  //////     {
851                  //////         AllUsagesCore(sequence[i], usages);
852                  //////     }
853                  //////     //for (int i = 0; i < matches[0].Count; i++)
854                  //////     //    AllUsagesCore(matches[0][i], usages);
855                  //////     //usages.UnionWith(matches[0]);
856                  //////     return usages.ToList();
857                  //////}
858                  var firstLinkUsages = new HashSet<ulong>();
859                  AllUsagesCore(sequence[0], firstLinkUsages);
860                  firstLinkUsages.Add(sequence[0]);
861                  //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
                     ↪  sequence[0] }; // or all sequences, containing this element?
862                  //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
                     ↪  1).ToList();
863                  var results = new HashSet<ulong>();
864                  foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
                     ↪  firstLinkUsages, 1))
865                  {
866                      AllUsagesCore(match, results);
867                  }
868                  return results.ToList();
869              }
870              return new List<ulong>();
871          });
872      }
873
874      /// <remarks>
875      /// TODO: Может потробоваться ограничение на уровень глубины рекурсии
876      /// </remarks>
877      public HashSet<ulong> AllUsages(ulong link)
878      {
879          return _sync.ExecuteReadOperation(() =>
880          {
881              var usages = new HashSet<ulong>();
882              AllUsagesCore(link, usages);
883              return usages;
884          });
885      }
886
887      // При сборе всех использований (последовательностей) можно сохранять  обратный путь к
         ↪  той связи с которой начинался поиск (STTTSSSTT),
888      // причём достаточно одного бита для хранения перехода влево или вправо
889      private void AllUsagesCore(ulong link, HashSet<ulong> usages)
890      {
891          bool handler(ulong doublet)
892          {
893              if (usages.Add(doublet))
894              {
895                  AllUsagesCore(doublet, usages);
896              }
897              return true;
898          }
899          Links.Unsync.Each(link, Constants.Any, handler);
900          Links.Unsync.Each(Constants.Any, link, handler);
901      }
902
903      public HashSet<ulong> AllBottomUsages(ulong link)
904      {
905          return _sync.ExecuteReadOperation(() =>
906          {
907              var visits = new HashSet<ulong>();
908              var usages = new HashSet<ulong>();
909              AllBottomUsagesCore(link, visits, usages);
910              return usages;
911          });
912      }
913
914      private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
         ↪  usages)
915      {
```

```csharp
            bool handler(ulong doublet)
            {
                if (visits.Add(doublet))
                {
                    AllBottomUsagesCore(doublet, visits, usages);
                }
                return true;
            }
            if (Links.Unsync.Count(Constants.Any, link) == 0)
            {
                usages.Add(link);
            }
            else
            {
                Links.Unsync.Each(link, Constants.Any, handler);
                Links.Unsync.Each(Constants.Any, link, handler);
            }
        }

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

        private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
            LinkIndex> outerHandler)
        {
            bool handler(ulong doublet)
            {
                if (usages.Add(doublet))
                {
                    if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
                    {
                        return false;
                    }
                    if (!AllUsagesCore1(doublet, usages, outerHandler))
                    {
                        return false;
                    }
                }
                return true;
            }
            return Links.Unsync.Each(link, Constants.Any, handler)
                && Links.Unsync.Each(Constants.Any, link, handler);
        }

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

        public void CalculateAllUsages2(ulong[] totals)
        {
            var calculator = new AllUsagesCalculator2(Links, totals);
            calculator.Calculate();
        }

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

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

```csharp
            public void Calculate() => _links.Each(_links.Constants.Any, _links.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, _links.Constants.Any, linkCalculator);
                    _links.Unsync.Each(_links.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(_links.Constants.Any, _links.Constants.Any,
            ↪  CalculateCore);

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

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

```
1068                        if (isElement(element))
1069                        {
1070                            if (stack.Count == 0)
1071                            {
1072                                break;
1073                            }
1074                            element = stack.Pop();
1075                            var source = getSource(element);
1076                            var target = getTarget(element);
1077                            // Обработка элемента
1078                            if (isElement(target))
1079                            {
1080                                visitLeaf(target);
1081                            }
1082                            if (isElement(source))
1083                            {
1084                                visitLeaf(source);
1085                            }
1086                            element = source;
1087                        }
1088                        else
1089                        {
1090                            stack.Push(element);
1091                            visitNode(element);
1092                            element = getTarget(element);
1093                        }
1094                    }
1095                }
1096                _totals[link]++;
1097                return true;
1098            }
1099        }
1100
1101        private class AllUsagesCollector
1102        {
1103            private readonly ILinks<ulong> _links;
1104            private readonly HashSet<ulong> _usages;
1105
1106            public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1107            {
1108                _links = links;
1109                _usages = usages;
1110            }
1111
1112            public bool Collect(ulong link)
1113            {
1114                if (_usages.Add(link))
1115                {
1116                    _links.Each(link, _links.Constants.Any, Collect);
1117                    _links.Each(_links.Constants.Any, link, Collect);
1118                }
1119                return true;
1120            }
1121        }
1122
1123        private class AllUsagesCollector1
1124        {
1125            private readonly ILinks<ulong> _links;
1126            private readonly HashSet<ulong> _usages;
1127            private readonly ulong _continue;
1128
1129            public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1130            {
1131                _links = links;
1132                _usages = usages;
1133                _continue = _links.Constants.Continue;
1134            }
1135
1136            public ulong Collect(IList<ulong> link)
1137            {
1138                var linkIndex = _links.GetIndex(link);
1139                if (_usages.Add(linkIndex))
1140                {
1141                    _links.Each(Collect, _links.Constants.Any, linkIndex);
1142                }
1143                return _continue;
1144            }
1145        }
1146
1147        private class AllUsagesCollector2
```

```csharp
        {
            private readonly ILinks<ulong> _links;
            private readonly BitString _usages;

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

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

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

        private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
        ↪ right)
        {
            // Direct
            if (left == right)
            {
                handler(new LinkAddress<LinkIndex>(left));
            }
            var doublet = Links.Unsync.SearchOrDefault(left, right);
            if (doublet != Constants.Null)
            {
                handler(new LinkAddress<LinkIndex>(doublet));
            }
            // Inner
            CloseInnerConnections(handler, left, right);
            // Outer
            StepLeft(handler, left, right);
            StepRight(handler, left, right);
            PartialStepRight(handler, left, right);
            PartialStepLeft(handler, left, right);
```

```csharp
            }

            private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
            ↪  HashSet<ulong> previousMatchings, long startAt)
            {
                if (startAt >= sequence.Length) // ?
                {
                    return previousMatchings;
                }
                var secondLinkUsages = new HashSet<ulong>();
                AllUsagesCore(sequence[startAt], secondLinkUsages);
                secondLinkUsages.Add(sequence[startAt]);
                var matchings = new HashSet<ulong>();
                var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
                //for (var i = 0; i < previousMatchings.Count; i++)
                foreach (var secondLinkUsage in secondLinkUsages)
                {
                    foreach (var previousMatching in previousMatchings)
                    {
                        //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
                        ↪  secondLinkUsage);
                        StepRight(filler.AddFirstAndReturnConstant, previousMatching,
                        ↪  secondLinkUsage);
                        TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
                        ↪  previousMatching);
                        //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
                        ↪  sequence[startAt]); // почему-то эта ошибочная запись приводит к
                        ↪  желаемым результам.
                        PartialStepRight(filler.AddFirstAndReturnConstant, 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] != links.Constants.Any && sequence[i] != ZeroOrMany &&
                    ↪  !links.Exists(sequence[i]))
                    {
                        throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                        ↪  $"patternSequence[{i}]");
                    }
                }
            }

            // Pattern Matching -> Key To Triggers
            public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
            {
                return _sync.ExecuteReadOperation(() =>
                {
                    patternSequence = Simplify(patternSequence);
                    if (patternSequence.Length > 0)
                    {
                        EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
                        var uniqueSequenceElements = new HashSet<ulong>();
                        for (var i = 0; i < patternSequence.Length; i++)
                        {
                            if (patternSequence[i] != Constants.Any && patternSequence[i] !=
                            ↪  ZeroOrMany)
                            {
                                uniqueSequenceElements.Add(patternSequence[i]);
                            }
                        }
                        var results = new HashSet<ulong>();
                        foreach (var uniqueSequenceElement in uniqueSequenceElements)
```

```
1290                         {
1291                             AllUsagesCore(uniqueSequenceElement, results);
1292                         }
1293                         var filteredResults = new HashSet<ulong>();
1294                         var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1295                         matcher.AddAllPatternMatchedToResults(results);
1296                         return filteredResults;
1297                     }
1298                     return new HashSet<ulong>();
1299                 });
1300         }
1301
1302         // Найти все возможные связи между указанным списком связей.
1303         // Находит связи между всеми указанными связями в любом порядке.
1304         // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1305         //     несколько раз в последовательности)
1305         public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1306         {
1307             return _sync.ExecuteReadOperation(() =>
1308             {
1309                 var results = new HashSet<ulong>();
1310                 if (linksToConnect.Length > 0)
1311                 {
1312                     Links.EnsureEachLinkExists(linksToConnect);
1313                     AllUsagesCore(linksToConnect[0], results);
1314                     for (var i = 1; i < linksToConnect.Length; i++)
1315                     {
1316                         var next = new HashSet<ulong>();
1317                         AllUsagesCore(linksToConnect[i], next);
1318                         results.IntersectWith(next);
1319                     }
1320                 }
1321                 return results;
1322             });
1323         }
1324
1325         public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1326         {
1327             return _sync.ExecuteReadOperation(() =>
1328             {
1329                 var results = new HashSet<ulong>();
1330                 if (linksToConnect.Length > 0)
1331                 {
1332                     Links.EnsureEachLinkExists(linksToConnect);
1333                     var collector1 = new AllUsagesCollector(Links.Unsync, results);
1334                     collector1.Collect(linksToConnect[0]);
1335                     var next = new HashSet<ulong>();
1336                     for (var i = 1; i < linksToConnect.Length; i++)
1337                     {
1338                         var collector = new AllUsagesCollector(Links.Unsync, next);
1339                         collector.Collect(linksToConnect[i]);
1340                         results.IntersectWith(next);
1341                         next.Clear();
1342                     }
1343                 }
1344                 return results;
1345             });
1346         }
1347
1348         public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1349         {
1350             return _sync.ExecuteReadOperation(() =>
1351             {
1352                 var results = new HashSet<ulong>();
1353                 if (linksToConnect.Length > 0)
1354                 {
1355                     Links.EnsureEachLinkExists(linksToConnect);
1356                     var collector1 = new AllUsagesCollector(Links, results);
1357                     collector1.Collect(linksToConnect[0]);
1358                     //AllUsagesCore(linksToConnect[0], results);
1359                     for (var i = 1; i < linksToConnect.Length; i++)
1360                     {
1361                         var next = new HashSet<ulong>();
1362                         var collector = new AllUsagesIntersectingCollector(Links, results, next);
1363                         collector.Collect(linksToConnect[i]);
1364                         //AllUsagesCore(linksToConnect[i], next);
1365                         //results.IntersectWith(next);
1366                         results = next;
```

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

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

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

```csharp
                   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);
                       }
                   }
               }
           }
           else
           {
               var nextRightLink = middleLinks[rightBound];
```

```
1518                    var elements = GetLeftElements(rightLink, nextRightLink);
1519                    if (leftBound <= rightBound)
1520                    {
1521                        for (var i = elements.Length - 1; i >= 0; i--)
1522                        {
1523                            var element = elements[i];
1524                            if (element != 0)
1525                            {
1526                                CollectMatchingSequences(leftLink, leftBound, middleLinks,
                                    ↪  elements[i], rightBound - 1, ref results);
1527                            }
1528                        }
1529                    }
1530                    else
1531                    {
1532                        for (var i = elements.Length - 1; i >= 0; i--)
1533                        {
1534                            var element = elements[i];
1535                            if (element != 0)
1536                            {
1537                                results.Add(element);
1538                            }
1539                        }
1540                    }
1541                }
1542            }
1543
1544        public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1545        {
1546            var result = new ulong[5];
1547            TryStepRight(startLink, rightLink, result, 0);
1548            Links.Each(Constants.Any, startLink, couple =>
1549            {
1550                if (couple != startLink)
1551                {
1552                    if (TryStepRight(couple, rightLink, result, 2))
1553                    {
1554                        return false;
1555                    }
1556                }
1557                return true;
1558            });
1559            if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1560            {
1561                result[4] = startLink;
1562            }
1563            return result;
1564        }
1565
1566        public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1567        {
1568            var added = 0;
1569            Links.Each(startLink, Constants.Any, couple =>
1570            {
1571                if (couple != startLink)
1572                {
1573                    var coupleTarget = Links.GetTarget(couple);
1574                    if (coupleTarget == rightLink)
1575                    {
1576                        result[offset] = couple;
1577                        if (++added == 2)
1578                        {
1579                            return false;
1580                        }
1581                    }
1582                    else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
                        ↪  == Net.And &&
1583                    {
1584                        result[offset + 1] = couple;
1585                        if (++added == 2)
1586                        {
1587                            return false;
1588                        }
1589                    }
1590                }
1591                return true;
1592            });
1593            return added > 0;
```

```csharp
            }

        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
            }

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

```
1673                    }
1674
1675            private readonly List<PatternBlock> _pattern;
1676            private int _patternPosition;
1677            private long _sequencePosition;
1678
1679            #endregion
1680
1681            public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
                ↪ HashSet<LinkIndex> results)
1682                : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1683            {
1684                _sequences = sequences;
1685                _patternSequence = patternSequence;
1686                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪ _sequences.Constants.Any && x != ZeroOrMany));
1687                _results = results;
1688                _pattern = CreateDetailedPattern();
1689            }
1690
1691            protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
                ↪ base.IsElement(link);
1692
1693            public bool PatternMatch(LinkIndex sequenceToMatch)
1694            {
1695                _patternPosition = 0;
1696                _sequencePosition = 0;
1697                foreach (var part in Walk(sequenceToMatch))
1698                {
1699                    if (!PatternMatchCore(part))
1700                    {
1701                        break;
1702                    }
1703                }
1704                return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
                    ↪ - 1 && _pattern[_patternPosition].Start == 0);
1705            }
1706
1707            private List<PatternBlock> CreateDetailedPattern()
1708            {
1709                var pattern = new List<PatternBlock>();
1710                var patternBlock = new PatternBlock();
1711                for (var i = 0; i < _patternSequence.Length; i++)
1712                {
1713                    if (patternBlock.Type == PatternBlockType.Undefined)
1714                    {
1715                        if (_patternSequence[i] == _sequences.Constants.Any)
1716                        {
1717                            patternBlock.Type = PatternBlockType.Gap;
1718                            patternBlock.Start = 1;
1719                            patternBlock.Stop = 1;
1720                        }
1721                        else if (_patternSequence[i] == ZeroOrMany)
1722                        {
1723                            patternBlock.Type = PatternBlockType.Gap;
1724                            patternBlock.Start = 0;
1725                            patternBlock.Stop = long.MaxValue;
1726                        }
1727                        else
1728                        {
1729                            patternBlock.Type = PatternBlockType.Elements;
1730                            patternBlock.Start = i;
1731                            patternBlock.Stop = i;
1732                        }
1733                    }
1734                    else if (patternBlock.Type == PatternBlockType.Elements)
1735                    {
1736                        if (_patternSequence[i] == _sequences.Constants.Any)
1737                        {
1738                            pattern.Add(patternBlock);
1739                            patternBlock = new PatternBlock
1740                            {
1741                                Type = PatternBlockType.Gap,
1742                                Start = 1,
1743                                Stop = 1
1744                            };
1745                        }
1746                        else if (_patternSequence[i] == ZeroOrMany)
1747                        {
1748                            pattern.Add(patternBlock);
```

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

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

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

        // matchstar: search for c*regexp at beginning of text
        //int matchstar(int c, char* regexp, char* text)
        //{
        //    do
        //    {    /* a * matches zero or more instances */
        //        if (matchhere(regexp, text))
        //            return 1;
        //    } while (*text != '\0' && (*text++ == c || c == '.'));
        //    return 0;
        //}
```

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

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

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

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

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

            #endregion
        }
    }
```

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

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

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

namespace Platform.Data.Doublets.Sequences
{
    public static class SequencesExtensions
    {
        public static TLink Create<TLink>(this ILinks<TLink> sequences, IList<TLink[]>
        ↪  groupedSequence)
        {
            var finalSequence = new TLink[groupedSequence.Count];
            for (var i = 0; i < finalSequence.Length; i++)
            {
                var part = groupedSequence[i];
                finalSequence[i] = part.Length == 1 ? part[0] :
                ↪  sequences.Create(part.ConvertToRestrictionsValues());
            }
            return sequences.Create(finalSequence.ConvertToRestrictionsValues());
```

```
19              }
20
21          public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
22          {
23              var list = new List<TLink>();
24              var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
25              sequences.Each(filler.AddAllValuesAndReturnConstant, new
                ↪  LinkAddress<TLink>(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          public bool ReadFullSequence { get; set; }
35
36          // TODO: Реализовать компактификацию при чтении
37          //public bool EnforceSingleSequenceVersionOnRead { get; set; }
38          //public bool UseRequestMarker { get; set; }
39          //public bool StoreRequestResults { get; set; }
40
41          public void InitOptions(ISynchronizedLinks<TLink> links)
42          {
43              if (UseSequenceMarker)
44              {
45                  if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
46                  {
47                      SequenceMarkerLink = links.CreatePoint();
48                  }
49                  else
50                  {
51                      if (!links.Exists(SequenceMarkerLink))
52                      {
53                          var link = links.CreatePoint();
54                          if (!_equalityComparer.Equals(link, SequenceMarkerLink))
55                          {
56                              throw new InvalidOperationException("Cannot recreate sequence marker
                                ↪  link.");
57                          }
58                      }
59                  }
60                  if (MarkedSequenceMatcher == null)
61                  {
62                      MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                        ↪  SequenceMarkerLink);
```

```
63                    }
64                }
65                var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
66                if (UseCompression)
67                {
68                    if (LinksToSequenceConverter == null)
69                    {
70                        ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
71                        if (UseSequenceMarker)
72                        {
73                            totalSequenceSymbolFrequencyCounter = new
                             ↪  TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                             ↪  MarkedSequenceMatcher);
74                        }
75                        else
76                        {
77                            totalSequenceSymbolFrequencyCounter = new
                             ↪  TotalSequenceSymbolFrequencyCounter<TLink>(links);
78                        }
79                        var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                         ↪  totalSequenceSymbolFrequencyCounter);
80                        var compressingConverter = new CompressingConverter<TLink>(links,
                         ↪  balancedVariantConverter, doubletFrequenciesCache);
81                        LinksToSequenceConverter = compressingConverter;
82                    }
83                }
84                else
85                {
86                    if (LinksToSequenceConverter == null)
87                    {
88                        LinksToSequenceConverter = balancedVariantConverter;
89                    }
90                }
91                if (UseIndex && Index == null)
92                {
93                    Index = new SequenceIndex<TLink>(links);
94                }
95                if (Walker == null)
96                {
97                    Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
98                }
99            }
100
101        public void ValidateOptions()
102        {
103            if (UseGarbageCollection && !UseSequenceMarker)
104            {
105                throw new NotSupportedException("To use garbage collection UseSequenceMarker
                 ↪  option must be on.");
106            }
107        }
108    }
109 }
```

./Platform.Data.Doublets/Sequences/SetFiller.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.Sequences
7   {
8       public class SetFiller<TElement, TReturnConstant>
9       {
10          protected readonly ISet<TElement> _set;
11          protected readonly TReturnConstant _returnConstant;
12
13          public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
14          {
15              _set = set;
16              _returnConstant = returnConstant;
17          }
18
19          public SetFiller(ISet<TElement> set) : this(set, default) { }
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          public void Add(TElement element) => _set.Add(element);
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
25          public bool AddAndReturnTrue(TElement element)
26          {
27              _set.Add(element);
28              return true;
29          }
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public bool AddFirstAndReturnTrue(IList<TElement> collection)
33          {
34              _set.Add(collection[0]);
35              return true;
36          }
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          public TReturnConstant AddAndReturnConstant(TElement element)
40          {
41              _set.Add(element);
42              return _returnConstant;
43          }
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          public TReturnConstant AddFirstAndReturnConstant(IList<TElement> collection)
47          {
48              _set.Add(collection[0]);
49              return _returnConstant;
50          }
51      }
52  }
```

./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;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Collections.Stacks;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences.Walkers
9   {
10      public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11      {
12          public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
            ↪ isElement) : base(links, stack, isElement) { }
13
14          public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
            ↪ links.IsPartialPoint) { }
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override TLink GetNextElementAfterPop(TLink element) =>
            ↪ Links.GetSource(element);
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override TLink GetNextElementAfterPush(TLink element) =>
            ↪ Links.GetTarget(element);
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected override IEnumerable<TLink> WalkContents(TLink element)
24          {
25              var parts = Links.GetLink(element);
26              var start = Links.Constants.IndexPart + 1;
27              for (var i = parts.Count - 1; i >= start; i--)
28              {
29                  var part = parts[i];
30                  if (IsElement(part))
31                  {
32                      yield return part;
```

```
33                          }
34                    }
35              }
36        }
37  }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  //#define USEARRAYPOOL
8  #if USEARRAYPOOL
9  using Platform.Collections;
10 #endif
11
12 namespace Platform.Data.Doublets.Sequences.Walkers
13 {
14     public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
15     {
16         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
17
18         private readonly Func<TLink, bool> _isElement;
19
20         public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
           ↪  base(links) => _isElement = isElement;
21
22         public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
           ↪  Links.IsPartialPoint;
23
24         public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
25
26         public TLink[] ToArray(TLink sequence)
27         {
28             var length = 1;
29             var array = new TLink[length];
30             array[0] = sequence;
31             if (_isElement(sequence))
32             {
33                 return array;
34             }
35             bool hasElements;
36             do
37             {
38                 length *= 2;
39 #if USEARRAYPOOL
40                 var nextArray = ArrayPool.Allocate<ulong>(length);
41 #else
42                 var nextArray = new TLink[length];
43 #endif
44                 hasElements = false;
45                 for (var i = 0; i < array.Length; i++)
46                 {
47                     var candidate = array[i];
48                     if (_equalityComparer.Equals(array[i], default))
49                     {
50                         continue;
51                     }
52                     var doubletOffset = i * 2;
53                     if (_isElement(candidate))
54                     {
55                         nextArray[doubletOffset] = candidate;
56                     }
57                     else
58                     {
59                         var link = Links.GetLink(candidate);
60                         var linkSource = Links.GetSource(link);
61                         var linkTarget = Links.GetTarget(link);
62                         nextArray[doubletOffset] = linkSource;
63                         nextArray[doubletOffset + 1] = linkTarget;
64                         if (!hasElements)
65                         {
66                             hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
67                         }
68                     }
69                 }
70 #if USEARRAYPOOL
```

```csharp
                        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;
            }
        }
    }
```

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

```csharp
using System;
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, Func<TLink, bool>
        ↪  isElement) : base(links, stack, isElement) { }

        public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
        ↪  stack, links.IsPartialPoint) { }

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

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

```
22            [MethodImpl(MethodImplOptions.AggressiveInlining)]
23            protected override IEnumerable<TLink> WalkContents(TLink element)
24            {
25                var parts = Links.GetLink(element);
26                for (var i = Links.Constants.IndexPart + 1; i < parts.Count; i++)
27                {
28                    var part = parts[i];
29                    if (IsElement(part))
30                    {
31                        yield return part;
32                    }
33                }
34            }
35        }
36    }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Collections.Stacks;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences.Walkers
9   {
10      public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
      ↪   ISequenceWalker<TLink>
11      {
12          private readonly IStack<TLink> _stack;
13          private readonly Func<TLink, bool> _isElement;
14
15          protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
      ↪   isElement) : base(links)
16          {
17              _stack = stack;
18              _isElement = isElement;
19          }
20
21          protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
      ↪   stack, links.IsPartialPoint)
22          {
23          }
24
25          public IEnumerable<TLink> Walk(TLink sequence)
26          {
27              _stack.Clear();
28              var element = sequence;
29              if (IsElement(element))
30              {
31                  yield return element;
32              }
33              else
34              {
35                  while (true)
36                  {
37                      if (IsElement(element))
38                      {
39                          if (_stack.IsEmpty)
40                          {
41                              break;
42                          }
43                          element = _stack.Pop();
44                          foreach (var output in WalkContents(element))
45                          {
46                              yield return output;
47                          }
48                          element = GetNextElementAfterPop(element);
49                      }
50                      else
51                      {
52                          _stack.Push(element);
53                          element = GetNextElementAfterPush(element);
54                      }
55                  }
56              }
57          }
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract IEnumerable<TLink> WalkContents(TLink element);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Stacks
{
    public class Stack<TLink> : IStack<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;

        private readonly ILinks<TLink> _links;
        private readonly TLink _stack;

        public bool IsEmpty => _equalityComparer.Equals(Peek(), _stack);

        public Stack(ILinks<TLink> links, TLink stack)
        {
            _links = links;
            _stack = stack;
        }

        private TLink GetStackMarker() => _links.GetSource(_stack);

        private TLink GetTop() => _links.GetTarget(_stack);

        public TLink Peek() => _links.GetTarget(GetTop());

        public TLink Pop()
        {
            var element = Peek();
            if (!_equalityComparer.Equals(element, _stack))
            {
                var top = GetTop();
                var previousTop = _links.GetSource(top);
                _links.Update(_stack, GetStackMarker(), previousTop);
                _links.Delete(top);
            }
            return element;
        }

        public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
            _links.GetOrCreate(GetTop(), element));
    }
}
```

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

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

namespace Platform.Data.Doublets.Stacks
{
    public static class StackExtensions
    {
        public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
        {
            var stackPoint = links.CreatePoint();
            var stack = links.Update(stackPoint, stackMarker, stackPoint);
            return stack;
        }
    }
}
```

./Platform.Data.Doublets/SynchronizedLinks.cs

```csharp
1  using System;
2  using System.Collections.Generic;
3  using Platform.Data.Doublets;
4  using Platform.Threading.Synchronization;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets
9  {
10     /// <remarks>
11     /// TODO: Autogeneration of synchronized wrapper (decorator).
12     /// TODO: Try to unfold code of each method using IL generation for performance improvements.
13     /// TODO: Or even to unfold multiple layers of implementations.
14     /// </remarks>
15     public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
16     {
17         public LinksConstants<TLinkAddress> Constants { get; }
18         public ISynchronization SyncRoot { get; }
19         public ILinks<TLinkAddress> Sync { get; }
20         public ILinks<TLinkAddress> Unsync { get; }
21
22         public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
            ↪ ReaderWriterLockSynchronization(), links) { }
23
24         public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
25         {
26             SyncRoot = synchronization;
27             Sync = this;
28             Unsync = links;
29             Constants = links.Constants;
30         }
31
32         public TLinkAddress Count(IList<TLinkAddress> restriction) =>
            ↪ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
33         public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
            ↪ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
            ↪ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
34         public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
            ↪ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
35         public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
            ↪ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
            ↪ Unsync.Update);
36         public void Delete(IList<TLinkAddress> restrictions) =>
            ↪ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
37
38         //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
            ↪ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
39         //{
40         //    if (restriction != null && substitution != null &&
            ↪ !substitution.EqualTo(restriction))
41         //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
            ↪ substitution, substitutedHandler, Unsync.Trigger);
42
43         //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
            ↪ substitutedHandler, Unsync.Trigger);
44         //}
45     }
46  }
```

./Platform.Data.Doublets/UInt64Link.cs

```csharp
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Platform.Exceptions;
5  using Platform.Ranges;
6  using Platform.Singletons;
7  using Platform.Collections.Lists;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets
12 {
13     /// <summary>
14     /// Структура описывающая уникальную связь.
15     /// </summary>
16     public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
17     {
```

```csharp
        private static readonly LinksConstants<ulong> _constants =
        ↪  Default<LinksConstants<ulong>>.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
            {
```

```csharp
                    Ensure.OnDebug.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.OnDebug.ArgumentNotNull(array, nameof(array));
                Ensure.OnDebug.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();

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

            #endregion
        }
    }
```

./Platform.Data.Doublets/UInt64LinkExtensions.cs

```csharp
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

```csharp
1   using System;
2   using System.Text;
3   using System.Collections.Generic;
4   using Platform.Singletons;
5   using Platform.Data.Exceptions;
6   using Platform.Data.Doublets.Unicode;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets
11  {
12      public static class UInt64LinksExtensions
13      {
14          public static readonly LinksConstants<ulong> Constants =
        ↪   Default<LinksConstants<ulong>>.Instance;
15
16          public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
17
18          public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
19          {
20              if (sequence == null)
21              {
22                  return;
23              }
24              for (var i = 0; i < sequence.Count; i++)
25              {
26                  if (!links.Exists(sequence[i]))
27                  {
28                      throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                ↪   $"sequence[{i}]");
29                  }
30              }
31          }
32
33          public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
        ↪   sequence)
34          {
35              if (sequence == null)
36              {
37                  return;
38              }
39              for (var i = 0; i < sequence.Count; i++)
40              {
41                  if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
42                  {
43                      throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                ↪   $"sequence[{i}]");
44                  }
45              }
46          }
47
48          public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
49          {
50              if (sequence == null)
51              {
52                  return false;
53              }
54              var constants = links.Constants;
55              for (var i = 0; i < sequence.Length; i++)
56              {
57                  if (sequence[i] == constants.Any)
58                  {
59                      return true;
60                  }
61              }
```

```csharp
                    return false;
                }

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

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

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

```
130                        else
131                        {
132                            links.AppendStructure(sb, visited, target.Index, isElement,
     ↪  appendElement, renderIndex);
133                        }
134                    }
135                    sb.Append(')');
136                }
137                else
138                {
139                    if (renderDebug)
140                    {
141                        sb.Append('*');
142                    }
143                    sb.Append(linkIndex);
144                }
145            }
146            else
147            {
148                if (renderDebug)
149                {
150                    sb.Append('~');
151                }
152                sb.Append(linkIndex);
153            }
154        }
155    }
156 }
```

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```
 1  using System;
 2  using System.Linq;
 3  using System.Collections.Generic;
 4  using System.IO;
 5  using System.Runtime.CompilerServices;
 6  using System.Threading;
 7  using System.Threading.Tasks;
 8  using Platform.Disposables;
 9  using Platform.Timestamps;
10  using Platform.Unsafe;
11  using Platform.IO;
12  using Platform.Data.Doublets.Decorators;
13  using Platform.Exceptions;
14
15  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17  namespace Platform.Data.Doublets
18  {
19      public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
20      {
21          /// <remarks>
22          /// Альтернативные варианты хранения трансформации (элемента транзакции):
23          ///
24          /// private enum TransitionType
25          /// {
26          ///     Creation,
27          ///     UpdateOf,
28          ///     UpdateTo,
29          ///     Deletion
30          /// }
31          ///
32          /// private struct Transition
33          /// {
34          ///     public ulong TransactionId;
35          ///     public UniqueTimestamp Timestamp;
36          ///     public TransactionItemType Type;
37          ///     public Link Source;
38          ///     public Link Linker;
39          ///     public Link Target;
40          /// }
41          ///
42          /// Или
43          ///
44          /// public struct TransitionHeader
45          /// {
46          ///     public ulong TransactionIdCombined;
47          ///     public ulong TimestampCombined;
48          ///
49          ///     public ulong TransactionId
```

```csharp
        ///     {
        ///         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.
        ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
        ↪ потребуется решить вопрос
        ///         со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
        ↪ пересечениями идентификаторов.
        ///
        /// Где хранить промежуточный список транзакций?
```

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

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

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

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

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

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

```csharp
                    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);

            public override ulong Create(IList<ulong> restrictions)
            {
                var createdLinkIndex = Links.Create();
                var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
                CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                    default, createdLink));
```

```csharp
274             return createdLinkIndex;
275         }
276
277         public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
278         {
279             var linkIndex = restrictions[Constants.IndexPart];
280             var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
281             linkIndex = Links.Update(restrictions, substitution);
282             var afterLink = new UInt64Link(Links.GetLink(linkIndex));
283             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
               ↪   beforeLink, afterLink));
284             return linkIndex;
285         }
286
287         public override void Delete(IList<ulong> restrictions)
288         {
289             var link = restrictions[Constants.IndexPart];
290             var deletedLink = new UInt64Link(Links.GetLink(link));
291             Links.Delete(link);
292             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
               ↪   deletedLink, default));
293         }
294
295         [MethodImpl(MethodImplOptions.AggressiveInlining)]
296         private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
               ↪   _transitions;
297
298         private void CommitTransition(Transition transition)
299         {
300             if (_currentTransaction != null)
301             {
302                 Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
303             }
304             var transitions = GetCurrentTransitions();
305             transitions.Enqueue(transition);
306         }
307
308         private void RevertTransition(Transition transition)
309         {
310             if (transition.After.IsNull()) // Revert Deletion with Creation
311             {
312                 Links.Create();
313             }
314             else if (transition.Before.IsNull()) // Revert Creation with Deletion
315             {
316                 Links.Delete(transition.After.Index);
317             }
318             else // Revert Update
319             {
320                 Links.Update(new[] { transition.After.Index, transition.Before.Source,
                   ↪   transition.Before.Target });
321             }
322         }
323
324         private void ResetCurrentTransation()
325         {
326             _currentTransactionId = 0;
327             _currentTransactionTransitions = null;
328             _currentTransaction = null;
329         }
330
331         private void PushTransitions()
332         {
333             if (_log == null || _transitions == null)
334             {
335                 return;
336             }
337             for (var i = 0; i < _transitions.Count; i++)
338             {
339                 var transition = _transitions.Dequeue();
340
341                 _log.Write(transition);
342                 _lastCommitedTransition = transition;
343             }
344         }
345
346         private void TransitionsPusher()
347         {
348             while (!IsDisposed && _transitionsPusher != null)
```

```
349             {
350                 Thread.Sleep(DefaultPushDelay);
351                 PushTransitions();
352             }
353         }
354
355         public Transaction BeginTransaction() => new Transaction(this);
356
357         private void DisposeTransitions()
358         {
359             try
360             {
361                 var pusher = _transitionsPusher;
362                 if (pusher != null)
363                 {
364                     _transitionsPusher = null;
365                     pusher.Wait();
366                 }
367                 if (_transitions != null)
368                 {
369                     PushTransitions();
370                 }
371                 _log.DisposeIfPossible();
372                 FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
373             }
374             catch (Exception ex)
375             {
376                 ex.Ignore();
377             }
378         }
379
380         #region DisposalBase
381
382         protected override void Dispose(bool manual, bool wasDisposed)
383         {
384             if (!wasDisposed)
385             {
386                 DisposeTransitions();
387             }
388             base.Dispose(manual, wasDisposed);
389         }
390
391         #endregion
392     }
393 }
```

./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  }
```

./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs

```
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 TLink[source.Length];
27               for (int i = 0; i < source.Length; i++)
28               {
29                   elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
30               }
31               _index.Add(elements);
32               var sequence = _listToSequenceLinkConverter.Convert(elements);
33               return Links.GetOrCreate(sequence, _unicodeSequenceMarker);
34           }
35       }
36   }
```

./Platform.Data.Doublets/Unicode/UnicodeMap.cs

```
1    using System;
2    using System.Collections.Generic;
3    using System.Globalization;
4    using System.Runtime.CompilerServices;
5    using System.Text;
6    using Platform.Data.Sequences;
7
8    #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               {
40                   _links.Delete(firstLink);
41               }
42               else
43               {
```

```csharp
                    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++)
            {
                linksSequence[i] = FromCharToLink(sequence[i]);
            }
            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
1  using Platform.Interfaces;
2  using System.Collections.Generic;
3
```

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

     namespace Platform.Data.Doublets.Unicode
     {
         public class UnicodeSequenceCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
         ↪ ICriterionMatcher<TLink>
         {
             private static readonly EqualityComparer<TLink> _equalityComparer =
             ↪ EqualityComparer<TLink>.Default;
             private readonly TLink _unicodeSequenceMarker;
             public UnicodeSequenceCriterionMatcher(ILinks<TLink> links, TLink unicodeSequenceMarker)
             ↪ : base(links) => _unicodeSequenceMarker = unicodeSequenceMarker;
             public bool IsMatched(TLink link) => _equalityComparer.Equals(Links.GetTarget(link),
             ↪ _unicodeSequenceMarker);
         }
     }
```

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

```csharp
using System;
using System.Linq;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<TLink, string>
    {
        private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
        private readonly ISequenceWalker<TLink> _sequenceWalker;
        private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;

        public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
        ↪ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
        ↪ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
        {
            _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
            _sequenceWalker = sequenceWalker;
            _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
        }

        public string Convert(TLink source)
        {
            if(!_unicodeSequenceCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                ↪ not a unicode sequence.");
            }
            var sequence = Links.GetSource(source);
            var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
            ↪ Convert).ToArray();
            return new string(charArray);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSymbolCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
    ↪ ICriterionMatcher<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;
        private readonly TLink _unicodeSymbolMarker;
        public UnicodeSymbolCriterionMatcher(ILinks<TLink> links, TLink unicodeSymbolMarker) :
        ↪ base(links) => _unicodeSymbolMarker = unicodeSymbolMarker;
        public bool IsMatched(TLink link) => _equalityComparer.Equals(Links.GetTarget(link),
        ↪ _unicodeSymbolMarker);
    }
}
```

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

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

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
        IConverter<TLink, char>
    {
        private readonly IConverter<TLink> _numberToAddressConverter;
        private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;

        public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
            numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
            base(links)
        {
            _numberToAddressConverter = numberToAddressConverter;
            _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
        }

        public char Convert(TLink source)
        {
            if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                    not a unicode symbol.");
            }
            return (char)(ushort)(Integer<TLink>)_numberToAddressConverter.Convert(Links.GetSour
                ce(source));
        }
    }
}
```

```csharp
using System;
using System.Collections.Generic;
using Xunit;
using Platform.Diagnostics;

namespace Platform.Data.Doublets.Tests
{
    public static class ComparisonTests
    {
        private class UInt64Comparer : IComparer<ulong>
        {
            public int Compare(ulong x, ulong y) => x.CompareTo(y);
        }

        private static int Compare(ulong x, ulong y) => x.CompareTo(y);

        [Fact]
        public static void GreaterOrEqualPerfomanceTest()
        {
            const int N = 1000000;

            ulong x = 10;
            ulong y = 500;

            bool result = false;

            var ts1 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Compare(x, y) >= 0;
                }
            });

            var comparer1 = Comparer<ulong>.Default;

            var ts2 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = comparer1.Compare(x, y) >= 0;
                }
            });
```

```
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/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);
```

```csharp
                    }
                });

                var equalityComparer1 = EqualityComparer<ulong>.Default;

                var ts4 = Performance.Measure(() =>
                {
                    for (int i = 0; i < N; i++)
                    {
                        result = equalityComparer1.Equals(x, y);
                    }
                });

                var equalityComparer2 = new UInt64EqualityComparer();

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

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

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

                var comparer = Comparer<ulong>.Default;

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

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

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

./Platform.Data.Doublets.Tests/GenericLinksTests.cs

```csharp
using System;
using Xunit;
using Platform.Reflection;
using Platform.Memory;
using Platform.Scopes;
using Platform.Data.Doublets.ResizableDirectMemory;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class GenericLinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using<byte>(links => links.TestCRUDOperations());
            Using<ushort>(links => links.TestCRUDOperations());
            Using<uint>(links => links.TestCRUDOperations());
            Using<ulong>(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            Using<byte>(links => links.TestRawNumbersCRUDOperations());
            Using<ushort>(links => links.TestRawNumbersCRUDOperations());
            Using<uint>(links => links.TestRawNumbersCRUDOperations());
```

```
27                      Using<ulong>(links => links.TestRawNumbersCRUDOperations());
28              }

30          [Fact]
31          public static void MultipleRandomCreationsAndDeletionsTest()
32          {
33              //if (!RuntimeInformation.IsOSPlatform(OSPlatform.Linux))
34              //{
35              //    Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution(
                ↪    ).TestMultipleRandomCreationsAndDeletions(16)); // Cannot use more because
                ↪    current implementation of tree cuts out 5 bits from the address space.
36              //    Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolutio
                ↪    n().TestMultipleRandomCreationsAndDeletions(100));
37              //    Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution(
                ↪    ).TestMultipleRandomCreationsAndDeletions(100));
38              //}
39              Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
                ↪    tMultipleRandomCreationsAndDeletions(100));
40          }

42          private static void Using<TLink>(Action<ILinks<TLink>> action)
43          {
44              //using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪    ResizableDirectMemoryLinks<TLink>>>())
45              //{
46              //    action(scope.Use<ILinks<TLink>>());
47              //}
48              using (var memory = new HeapResizableDirectMemory())
49              {
50                  Unsafe.MemoryBlock.Zero((void*)memory.Pointer, memory.ReservedCapacity); // Bug
                ↪    workaround
51                  using (var links = new ResizableDirectMemoryLinks<TLink>(memory))
52                  {
53                      action(links);
54                  }
55              }
56          }
57      }
58  }
```

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

```csharp
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;

16  namespace Platform.Data.Doublets.Tests
17  {
18      public static class OptimalVariantSequenceTests
19      {
20          private const string SequenceExample = "зеленела зелёная зелень";

22          [Fact]
23          public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
24          {
25              using (var scope = new TempLinksTestScope(useSequences: false))
26              {
27                  var links = scope.Links;
28                  var constants = links.Constants;

30                  links.UseUnicode();

32                  var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);

34                  var meaningRoot = links.CreatePoint();
35                  var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
36                  var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
37                  var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                ↪    constants.Itself);
38
```

```csharp
39              var unaryNumberToAddressConverter = new
        ↪   UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
40              var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
41              var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
        ↪   frequencyMarker, unaryOne, unaryNumberIncrementer);
42              var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
        ↪   frequencyPropertyMarker, frequencyMarker);
43              var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
        ↪   frequencyPropertyOperator, frequencyIncrementer);
44              var linkToItsFrequencyNumberConverter = new
        ↪   LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
        ↪   unaryNumberToAddressConverter);
45              var sequenceToItsLocalElementLevelsConverter = new
        ↪   SequenceToItsLocalElementLevelsConverter<ulong>(links,
        ↪   linkToItsFrequencyNumberConverter);
46              var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
        ↪   sequenceToItsLocalElementLevelsConverter);
47
48              var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
        ↪   Walker = new LeveledSequenceWalker<ulong>(links) });
49
50              ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
        ↪   index, optimalVariantConverter);
51          }
52      }
53
54      [Fact]
55      public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
56      {
57          using (var scope = new TempLinksTestScope(useSequences: false))
58          {
59              var links = scope.Links;
60
61              links.UseUnicode();
62
63              var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
64
65              var linksToFrequencies = new Dictionary<ulong, ulong>();
66
67              var totalSequenceSymbolFrequencyCounter = new
        ↪   TotalSequenceSymbolFrequencyCounter<ulong>(links);
68
69              var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
        ↪   totalSequenceSymbolFrequencyCounter);
70
71              var index = new
        ↪   CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
72              var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
        ↪   ncyNumberConverter<ulong>(linkFrequenciesCache);
73
74              var sequenceToItsLocalElementLevelsConverter = new
        ↪   SequenceToItsLocalElementLevelsConverter<ulong>(links,
        ↪   linkToItsFrequencyNumberConverter);
75              var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
        ↪   sequenceToItsLocalElementLevelsConverter);
76
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

```
 1   using System.IO;
 2   using Xunit;
 3   using Platform.Singletons;
 4   using Platform.Memory;
 5   using Platform.Data.Doublets.ResizableDirectMemory;
 6
 7   namespace Platform.Data.Doublets.Tests
 8   {
 9       public static class ResizableDirectMemoryLinksTests
10       {
```

```
11        private static readonly LinksConstants<ulong> _constants =
    ↪    Default<LinksConstants<ulong>>.Instance;
12
13        [Fact]
14        public static void BasicFileMappedMemoryTest()
15        {
16            var tempFilename = Path.GetTempFileName();
17            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
18            {
19                memoryAdapter.TestBasicMemoryOperations();
20            }
21            File.Delete(tempFilename);
22        }
23
24        [Fact]
25        public static void BasicHeapMemoryTest()
26        {
27            using (var memory = new
    ↪    HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
28            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
    ↪    UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
29            {
30                memoryAdapter.TestBasicMemoryOperations();
31            }
32        }
33
34        private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
35        {
36            var link = memoryAdapter.Create();
37            memoryAdapter.Delete(link);
38        }
39
40        [Fact]
41        public static void NonexistentReferencesHeapMemoryTest()
42        {
43            using (var memory = new
    ↪    HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
44            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
    ↪    UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
45            {
46                memoryAdapter.TestNonexistentReferences();
47            }
48        }
49
50        private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
51        {
52            var link = memoryAdapter.Create();
53            memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
54            var resultLink = _constants.Null;
55            memoryAdapter.Each(foundLink =>
56            {
57                resultLink = foundLink[_constants.IndexPart];
58                return _constants.Break;
59            }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
60            Assert.True(resultLink == link);
61            Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
62            memoryAdapter.Delete(link);
63        }
64    }
65 }
```

./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  using Platform.Reflection;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public static class ScopeTests
11     {
12        [Fact]
13        public static void SingleDependencyTest()
14        {
15            using (var scope = new Scope())
16            {
17                scope.IncludeAssemblyOf<IMemory>();
```

```
18              var instance = scope.Use<IDirectMemory>();
19              Assert.IsType<HeapResizableDirectMemory>(instance);
20          }
21      }

23      [Fact]
24      public static void CascadeDependencyTest()
25      {
26          using (var scope = new Scope())
27          {
28              scope.Include<TemporaryFileMappedResizableDirectMemory>();
29              scope.Include<UInt64ResizableDirectMemoryLinks>();
30              var instance = scope.Use<ILinks<ulong>>();
31              Assert.IsType<UInt64ResizableDirectMemoryLinks>(instance);
32          }
33      }

35      [Fact]
36      public static void FullAutoResolutionTest()
37      {
38          using (var scope = new Scope(autoInclude: true, autoExplore: true))
39          {
40              var instance = scope.Use<UInt64Links>();
41              Assert.IsType<UInt64Links>(instance);
42          }
43      }

45      [Fact]
46      public static void TypeParametersTest()
47      {
48          using (var scope = new Scope<Types<HeapResizableDirectMemory,
     ↪  ResizableDirectMemoryLinks<ulong>>>())
49          {
50              var links = scope.Use<ILinks<ulong>>();
51              Assert.IsType<ResizableDirectMemoryLinks<ulong>>(links);
52          }
53      }
54  }
55 }
```

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

```csharp
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.Doublets.Sequences;
11 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
12 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
13 using Platform.Data.Doublets.Sequences.Converters;
14 using Platform.Data.Doublets.Unicode;

16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class SequencesTests
19     {
20         private static readonly LinksConstants<ulong> _constants =
     ↪  Default<LinksConstants<ulong>>.Instance;

22         static SequencesTests()
23         {
24             // Trigger static constructor to not mess with perfomance measurements
25             _ = BitString.GetBitMaskFromIndex(1);
26         }

28         [Fact]
29         public static void CreateAllVariantsTest()
30         {
31             const long sequenceLength = 8;

33             using (var scope = new TempLinksTestScope(useSequences: true))
34             {
35                 var links = scope.Links;
36                 var sequences = scope.Sequences;

38                 var sequence = new ulong[sequenceLength];
```

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

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

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

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

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

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

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

        //    const long sequenceLength = 8;

        //    const ulong itself = LinksConstants.Itself;

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

        //        SequencesOptions o = new SequencesOptions();

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

        //        var sequences = new Sequences(links);

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

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

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

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

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

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

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

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

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

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

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

```
269              var sequence = new ulong[sequenceLength];
270              for (var i = 0; i < sequenceLength; i++)
271              {
272                  sequence[i] = links.Create();
273              }
274
275              var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
276
277              var balancedVariant = balancedVariantConverter.Convert(sequence);
278
279              var partialSequence = new ulong[sequenceLength - 2];
280
281              Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
282
283              var sw1 = Stopwatch.StartNew();
284              var searchResults1 =
      ↪      sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
285
286              var sw2 = Stopwatch.StartNew();
287              var searchResults2 =
      ↪      sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
288
289              Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
290
291              Assert.True(searchResults2.Count == 1 && balancedVariant ==
      ↪      searchResults2.First());
292
293              for (var i = 0; i < sequenceLength; i++)
294              {
295                  links.Delete(sequence[i]);
296              }
297          }
298      }
299
300      [Fact(Skip = "Correct implementation is pending")]
301      public static void PatternMatchTest()
302      {
303          var zeroOrMany = Sequences.Sequences.ZeroOrMany;
304
305          using (var scope = new TempLinksTestScope(useSequences: true))
306          {
307              var links = scope.Links;
308              var sequences = scope.Sequences;
309
310              var e1 = links.Create();
311              var e2 = links.Create();
312
313              var sequence = new[]
314              {
315                  e1, e2, e1, e2 // mama / papa
316              };
317
318              var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
319
320              var balancedVariant = balancedVariantConverter.Convert(sequence);
321
322              // 1: [1]
323              // 2: [2]
324              // 3: [1,2]
325              // 4: [1,2,1,2]
326
327              var doublet = links.GetSource(balancedVariant);
328
329              var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
330
331              Assert.True(matchedSequences1.Count == 0);
332
333              var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
334
335              Assert.True(matchedSequences2.Count == 0);
336
337              var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
338
339              Assert.True(matchedSequences3.Count == 0);
340
341              var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
342
343              Assert.Contains(doublet, matchedSequences4);
344              Assert.Contains(balancedVariant, matchedSequences4);
345
```

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

        [Fact]
        public static void IndexTest()
        {
            using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
                true }, useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;
                var index = sequences.Options.Index;

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

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

                Assert.False(index.MightContain(sequence));

                index.Add(sequence);

                Assert.True(index.MightContain(sequence));
            }
        }

        /// <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>
        private static readonly string _exampleText =
            @"([english
                version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
            {
                var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
                ↪ StringSplitOptions.RemoveEmptyEntries);
                var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
                var totalCharacters = arrays.Select(x => x.Length).Sum();

            using (var scope1 = new TempLinksTestScope(useSequences: true))
            using (var scope2 = new TempLinksTestScope(useSequences: true))
            using (var scope3 = new TempLinksTestScope(useSequences: true))
            {
                scope1.Links.Unsync.UseUnicode();
                scope2.Links.Unsync.UseUnicode();
                scope3.Links.Unsync.UseUnicode();

                var balancedVariantConverter1 = new
                ↪ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
                var totalSequenceSymbolFrequencyCounter = new
                ↪ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
                var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
                ↪ totalSequenceSymbolFrequencyCounter);
                var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
                ↪ balancedVariantConverter1, linkFrequenciesCache1,
                ↪ doInitialFrequenciesIncrement: false);

                //var compressor2 = scope2.Sequences;
                var compressor3 = scope3.Sequences;

                var constants = Default<LinksConstants<ulong>>.Instance;

                var sequences = compressor3;
                //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 FrequencyPropertyOperator<ulong>(links,
                ↪ frequencyPropertyMarker, frequencyMarker);
                //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                ↪ frequencyPropertyOperator, frequencyIncrementer);
                //var linkToItsFrequencyNumberConverter = new
                ↪ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                ↪ unaryNumberToAddressConverter);

                var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
                ↪ totalSequenceSymbolFrequencyCounter);

                var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque⌋
                ↪ ncyNumberConverter<ulong>(linkFrequenciesCache3);

                var sequenceToItsLocalElementLevelsConverter = new
                ↪ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
                ↪ linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new
                ↪ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
                ↪ sequenceToItsLocalElementLevelsConverter);

                var compressed1 = new ulong[arrays.Length];
                var compressed2 = new ulong[arrays.Length];
                var compressed3 = new ulong[arrays.Length];

                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++)
```

```
547                      {
548                          linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
549                          compressed1[i] = compressor1.Convert(arrays[i]);
550                      }
551
552                  var elapsed1 = sw1.Elapsed;
553
554                  var balancedVariantConverter2 = new
      ↪   BalancedVariantConverter<ulong>(scope2.Links.Unsync);
555
556                  var initialCount2 = scope2.Links.Unsync.Count();
557
558                  var sw2 = Stopwatch.StartNew();
559
560                  for (int i = START; i < END; i++)
561                  {
562                      compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
563                  }
564
565                  var elapsed2 = sw2.Elapsed;
566
567                  for (int i = START; i < END; i++)
568                  {
569                      linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
570                  }
571
572                  var initialCount3 = scope3.Links.Unsync.Count();
573
574                  var sw3 = Stopwatch.StartNew();
575
576                  for (int i = START; i < END; i++)
577                  {
578                      //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
579                      compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
580                  }
581
582                  var elapsed3 = sw3.Elapsed;
583
584                  Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
      ↪   Optimal variant: {elapsed3}");
585
586                  // Assert.True(elapsed1 > elapsed2);
587
588                  // Checks
589                  for (int i = START; i < END; i++)
590                  {
591                      var sequence1 = compressed1[i];
592                      var sequence2 = compressed2[i];
593                      var sequence3 = compressed3[i];
594
595                      var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
      ↪   scope1.Links.Unsync);
596
597                      var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
      ↪   scope2.Links.Unsync);
598
599                      var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
      ↪   scope3.Links.Unsync);
600
601                      var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
      ↪   link.IsPartialPoint());
602                      var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
      ↪   link.IsPartialPoint());
603                      var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
      ↪   link.IsPartialPoint());
604
605                      //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
      ↪   arrays[i].Length > 3)
606                      //    Assert.False(structure1 == structure2);
607                      //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
      ↪   arrays[i].Length > 3)
608                      //    Assert.False(structure3 == structure2);
609
610                      Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
611                      Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
612                  }
613
614                  Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
      ↪   totalCharacters);
```

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

```csharp
            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].ConvertToRestrictionsValues());
                var second = compressor1.Create(arrays[i].ConvertToRestrictionsValues());

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

            var elapsed1 = sw1.Elapsed;

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

            var sw2 = Stopwatch.StartNew();

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

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

            var elapsed2 = sw2.Elapsed;

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

            Assert.True(elapsed1 > elapsed2);

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

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

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

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

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

```csharp
                        //    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;

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

                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]);
                }
```

```csharp
                    var elapsed2 = sw2.Elapsed;

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

                    Assert.True(elapsed1 > elapsed2);

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

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

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

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

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

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

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

                    //compressor1.ValidateFrequencies();
                }
            }

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

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

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

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

                var createResults = sequences.CreateAllVariants2(sequence);

                Global.Trash = createResults;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

                var createResults = sequences.CreateAllVariants2(sequence);

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

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

                    sequences.CalculateAllUsages(linksTotalUsages1);

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

                    sequences.CalculateAllUsages2(linksTotalUsages2);
```

```
983
984                    var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
985                    Assert.True(intersection1.Count == linksTotalUsages2.Length);
986                }
987
988                for (var i = 0; i < sequenceLength; i++)
989                {
990                    links.Delete(sequence[i]);
991                }
992            }
993        }
994    }
995 }
```

./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 ILinks<ulong> MemoryAdapter { get; }
12          public SynchronizedLinks<ulong> Links { get; }
13          public Sequences.Sequences Sequences { get; }
14          public string TempFilename { get; }
15          public string TempTransactionLogFilename { get; }
16          private readonly bool _deleteFiles;
17
18          public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
            ↪  useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
            ↪  useLog) { }
19
20          public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
            ↪  true, bool useSequences = false, bool useLog = false)
21          {
22              _deleteFiles = deleteFiles;
23              TempFilename = Path.GetTempFileName();
24              TempTransactionLogFilename = Path.GetTempFileName();
25              var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
26              MemoryAdapter = useLog ? (ILinks<ulong>)new
                  ↪  UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
                  ↪  coreMemoryAdapter;
27              Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
28              if (useSequences)
29              {
30                  Sequences = new Sequences.Sequences(Links, sequencesOptions);
31              }
32          }
33
34          protected override void Dispose(bool manual, bool wasDisposed)
35          {
36              if (!wasDisposed)
37              {
38                  Links.Unsync.DisposeIfPossible();
39                  if (_deleteFiles)
40                  {
41                      DeleteFiles();
42                  }
43              }
44          }
45
46          public void DeleteFiles()
47          {
48              File.Delete(TempFilename);
49              File.Delete(TempTransactionLogFilename);
50          }
51      }
52 }
```

./Platform.Data.Doublets.Tests/TestExtensions.cs

```
1   using System.Collections.Generic;
2   using Xunit;
3   using Platform.Ranges;
4   using Platform.Numbers;
5   using Platform.Random;
```

```csharp
using Platform.Setters;

namespace Platform.Data.Doublets.Tests
{
    public static class TestExtensions
    {
        public static void TestCRUDOperations<T>(this ILinks<T> links)
        {
            var constants = links.Constants;

            var equalityComparer = EqualityComparer<T>.Default;

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

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

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

            var linkAddress = links.Create();

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

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

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

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

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

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

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

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

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

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

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

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

            // Delete link
            links.Delete(linkAddress);

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

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

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

        public 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);
```

```
86          // Create Link (External -> External)
87          var linkAddress1 = links.Create();
88
89          links.Update(linkAddress1, h106E, h108E);
90
91          var link1 = new Link<T>(links.GetLink(linkAddress1));
92
93          Assert.True(equalityComparer.Equals(link1.Source, h106E));
94          Assert.True(equalityComparer.Equals(link1.Target, h108E));
95
96          // Create Link (Internal -> External)
97          var linkAddress2 = links.Create();
98
99          links.Update(linkAddress2, linkAddress1, h108E);
100
101          var link2 = new Link<T>(links.GetLink(linkAddress2));
102
103          Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
104          Assert.True(equalityComparer.Equals(link2.Target, h108E));
105
106          // Create Link (Internal -> Internal)
107          var linkAddress3 = links.Create();
108
109          links.Update(linkAddress3, linkAddress1, linkAddress2);
110
111          var link3 = new Link<T>(links.GetLink(linkAddress3));
112
113          Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
114          Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
115
116          // Search for created link
117          var setter1 = new Setter<T>(constants.Null);
118          links.Each(h106E, h108E, setter1.SetAndReturnFalse);
119
120          Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
121
122          // Search for nonexistent link
123          var setter2 = new Setter<T>(constants.Null);
124          links.Each(h106E, h107E, setter2.SetAndReturnFalse);
125
126          Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
127
128          // Update link to reference null (prepare for delete)
129          var updated = links.Update(linkAddress3, constants.Null, constants.Null);
130
131          Assert.True(equalityComparer.Equals(updated, linkAddress3));
132
133          link3 = new Link<T>(links.GetLink(linkAddress3));
134
135          Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
136          Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
137
138          // Delete link
139          links.Delete(linkAddress3);
140
141          Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));
142
143          var setter3 = new Setter<T>(constants.Null);
144          links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
145
146          Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
147      }
148
149      public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
    ↪   links, int maximumOperationsPerCycle)
150      {
151          var comparer = Comparer<TLink>.Default;
152          for (var N = 1; N < maximumOperationsPerCycle; N++)
153          {
154              var random = new System.Random(N);
155              var created = 0;
156              var deleted = 0;
157              for (var i = 0; i < N; i++)
158              {
159                  long linksCount = (Integer<TLink>)links.Count();
160                  var createPoint = random.NextBoolean();
161                  if (linksCount > 2 && createPoint)
162                  {
163                      var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
164                      TLink source = (Integer<TLink>)random.NextUInt64(linksAddressRange);
```

```
165            TLink target = (Integer<TLink>)random.NextUInt64(linksAddressRange);
         ↪  //-V3086
166            var resultLink = links.CreateAndUpdate(source, target);
167            if (comparer.Compare(resultLink, (Integer<TLink>)linksCount) > 0)
168            {
169                created++;
170            }
171        }
172        else
173        {
174            links.Create();
175            created++;
176        }
177    }
178    Assert.True(created == (Integer<TLink>)links.Count());
179    for (var i = 0; i < N; i++)
180    {
181        TLink link = (Integer<TLink>)(i + 1);
182        if (links.Exists(link))
183        {
184            links.Delete(link);
185            deleted++;
186        }
187    }
188    Assert.True((Integer<TLink>)links.Count() == 0);
189    }
190    }
191    }
192 }
```

./Platform.Data.Doublets.Tests/UInt64LinksTests.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.Doublets.ResizableDirectMemory;
18 using Platform.Data.Doublets.Decorators;
19
20 namespace Platform.Data.Doublets.Tests
21 {
22     public static class UInt64LinksTests
23     {
24         private static readonly LinksConstants<ulong> _constants =
           ↪  Default<LinksConstants<ulong>>.Instance;
25
26         private const long Iterations = 10 * 1024;
27
28         #region Concept
29
30         [Fact]
31         public static void MultipleCreateAndDeleteTest()
32         {
33             using (var scope = new TempLinksTestScope())
34             {
35                 scope.Links.TestMultipleRandomCreationsAndDeletions(100);
36             }
37         }
38
39         [Fact]
40         public static void CascadeUpdateTest()
41         {
42             var itself = _constants.Itself;
43
44             using (var scope = new TempLinksTestScope(useLog: true))
45             {
46                 var links = scope.Links;
47
48                 var l1 = links.Create();
49                 var l2 = links.Create();
```

```
50
51                    l2 = links.Update(l2, l2, l1, l2);
52
53                    links.CreateAndUpdate(l2, itself);
54                    links.CreateAndUpdate(l2, itself);
55
56                    l2 = links.Update(l2, l1);
57
58                    links.Delete(l2);
59
60                    Global.Trash = links.Count();
61
62                    links.Unsync.DisposeIfPossible(); // Close links to access log
63
64                    Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
       ↪    e.TempTransactionLogFilename);
65                }
66            }
67
68            [Fact]
69            public static void BasicTransactionLogTest()
70            {
71                using (var scope = new TempLinksTestScope(useLog: true))
72                {
73                    var links = scope.Links;
74                    var l1 = links.Create();
75                    var l2 = links.Create();
76
77                    Global.Trash = links.Update(l2, l2, l1, l2);
78
79                    links.Delete(l1);
80
81                    links.Unsync.DisposeIfPossible(); // Close links to access log
82
83                    Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
       ↪    e.TempTransactionLogFilename);
84                }
85            }
86
87            [Fact]
88            public static void TransactionAutoRevertedTest()
89            {
90                // Auto Reverted (Because no commit at transaction)
91                using (var scope = new TempLinksTestScope(useLog: true))
92                {
93                    var links = scope.Links;
94                    var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
95                    using (var transaction = transactionsLayer.BeginTransaction())
96                    {
97                        var l1 = links.Create();
98                        var l2 = links.Create();
99
100                       links.Update(l2, l2, l1, l2);
101                   }
102
103                   Assert.Equal(0UL, links.Count());
104
105                   links.Unsync.DisposeIfPossible();
106
107                   var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
       ↪    cope.TempTransactionLogFilename);
108                   Assert.Single(transitions);
109                }
110            }
111
112            [Fact]
113            public static void TransactionUserCodeErrorNoDataSavedTest()
114            {
115                // User Code Error (Autoreverted), no data saved
116                var itself = _constants.Itself;
117
118                TempLinksTestScope lastScope = null;
119                try
120                {
121                    using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
       ↪    useLog: true))
122                    {
123                        var links = scope.Links;
```

```csharp
124                             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
      ↪    atorBase<ulong>)links.Unsync).Links;
125                             using (var transaction = transactionsLayer.BeginTransaction())
126                             {
127                                 var l1 = links.CreateAndUpdate(itself, itself);
128                                 var l2 = links.CreateAndUpdate(itself, itself);

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

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

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

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

139                                 links.Delete(l2);

141                                 ExceptionThrower();

143                                 transaction.Commit();
144                             }

146                             Global.Trash = links.Count();
147                         }
148                     }
149                     catch
150                     {
151                         Assert.False(lastScope == null);

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

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

157                         lastScope.DeleteFiles();
158                     }
159                 }

161             [Fact]
162             public static void TransactionUserCodeErrorSomeDataSavedTest()
163             {
164                 // User Code Error (Autoreverted), some data saved
165                 var itself = _constants.Itself;

167                 TempLinksTestScope lastScope = null;
168                 try
169                 {
170                     ulong l1;
171                     ulong l2;

173                     using (var scope = new TempLinksTestScope(useLog: true))
174                     {
175                         var links = scope.Links;
176                         l1 = links.CreateAndUpdate(itself, itself);
177                         l2 = links.CreateAndUpdate(itself, itself);

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

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

184                         links.Unsync.DisposeIfPossible();

186                         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
      ↪    scope.TempTransactionLogFilename);
187                     }

189                     using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
      ↪    useLog: true))
190                     {
191                         var links = scope.Links;
192                         var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
193                         using (var transaction = transactionsLayer.BeginTransaction())
194                         {
195                             l2 = links.Update(l2, l1);
196
```

```
197                    links.Delete(l2);
198
199                    ExceptionThrower();
200
201                    transaction.Commit();
202                }
203
204                Global.Trash = links.Count();
205            }
206        }
207        catch
208        {
209            Assert.False(lastScope == null);
210
211            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
         ↪  Scope.TempTransactionLogFilename);
212
213            lastScope.DeleteFiles();
214        }
215    }
216
217    [Fact]
218    public static void TransactionCommit()
219    {
220        var itself = _constants.Itself;
221
222        var tempDatabaseFilename = Path.GetTempFileName();
223        var tempTransactionLogFilename = Path.GetTempFileName();
224
225        // Commit
226        using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
         ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
         ↪  tempTransactionLogFilename))
227        using (var links = new UInt64Links(memoryAdapter))
228        {
229            using (var transaction = memoryAdapter.BeginTransaction())
230            {
231                var l1 = links.CreateAndUpdate(itself, itself);
232                var l2 = links.CreateAndUpdate(itself, itself);
233
234                Global.Trash = links.Update(l2, l2, l1, l2);
235
236                links.Delete(l1);
237
238                transaction.Commit();
239            }
240
241            Global.Trash = links.Count();
242        }
243
244        Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
         ↪  sactionLogFilename);
245    }
246
247    [Fact]
248    public static void TransactionDamage()
249    {
250        var itself = _constants.Itself;
251
252        var tempDatabaseFilename = Path.GetTempFileName();
253        var tempTransactionLogFilename = Path.GetTempFileName();
254
255        // Commit
256        using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
         ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
         ↪  tempTransactionLogFilename))
257        using (var links = new UInt64Links(memoryAdapter))
258        {
259            using (var transaction = memoryAdapter.BeginTransaction())
260            {
261                var l1 = links.CreateAndUpdate(itself, itself);
262                var l2 = links.CreateAndUpdate(itself, itself);
263
264                Global.Trash = links.Update(l2, l2, l1, l2);
265
266                links.Delete(l1);
267
268                transaction.Commit();
269            }
```

```
270
271                     Global.Trash = links.Count();
272             }
273
274         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran↲
          ↪  sactionLogFilename);
275
276         // Damage database
277
278         FileHelpers.WriteFirst(tempTransactionLogFilename, new
          ↪  UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
279
280         // Try load damaged database
281         try
282         {
283             // TODO: Fix
284             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
              ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
              ↪  tempTransactionLogFilename))
285             using (var links = new UInt64Links(memoryAdapter))
286             {
287                 Global.Trash = links.Count();
288             }
289         }
290         catch (NotSupportedException ex)
291         {
292             Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
                ↪  yet.");
293         }
294
295         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran↲
          ↪  sactionLogFilename);
296
297         File.Delete(tempDatabaseFilename);
298         File.Delete(tempTransactionLogFilename);
299     }
300
301     [Fact]
302     public static void Bug1Test()
303     {
304         var tempDatabaseFilename = Path.GetTempFileName();
305         var tempTransactionLogFilename = Path.GetTempFileName();
306
307         var itself = _constants.Itself;
308
309         // User Code Error (Autoreverted), some data saved
310         try
311         {
312             ulong l1;
313             ulong l2;
314
315             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
              ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
              ↪  tempTransactionLogFilename))
316             using (var links = new UInt64Links(memoryAdapter))
317             {
318                 l1 = links.CreateAndUpdate(itself, itself);
319                 l2 = links.CreateAndUpdate(itself, itself);
320
321                 l2 = links.Update(l2, l2, l1, l2);
322
323                 links.CreateAndUpdate(l2, itself);
324                 links.CreateAndUpdate(l2, itself);
325             }
326
327             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp↲
              ↪  TransactionLogFilename);
328
329             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
              ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
              ↪  tempTransactionLogFilename))
330             using (var links = new UInt64Links(memoryAdapter))
331             {
332                 using (var transaction = memoryAdapter.BeginTransaction())
333                 {
334                     l2 = links.Update(l2, l1);
335
336                     links.Delete(l2);
```

```csharp
                    ExceptionThrower();

                    transaction.Commit();
                }

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

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

    private static void ExceptionThrower() => throw new InvalidOperationException();

    [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}}");
        }
```

```csharp
        }

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

        #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);
```

```
485         ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
486
487         var loops = totalLinksToCreate / linksStep;
488
489         for (int i = 0; i < loops; i++)
490         {
491             creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
492             searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));
493
494             Console.Write("\rC + S {0}/{1}", i + 1, loops);
495         }
496
497         ConsoleHelpers.Debug();
498
499         for (int i = 0; i < loops; i++)
500         {
501             deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
502
503             Console.Write("\rD {0}/{1}", i + 1, loops);
504         }
505
506         ConsoleHelpers.Debug();
507
508         ConsoleHelpers.Debug("C S D");
509
510         for (int i = 0; i < loops; i++)
511         {
512             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
     searchMeasuremets[i], deletionMeasurements[i]);
513         }
514
515         ConsoleHelpers.Debug("C S D (no overhead)");
516
517         for (int i = 0; i < loops; i++)
518         {
519             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
     searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
520         }
521
522         ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
     links.Total);
523     }
524
525     private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
     amountToCreate)
526     {
527         for (long i = 0; i < amountToCreate; i++)
528             links.Create(0, 0);
529     }
530
531     private static TimeSpan GetBaseRandomLoopOverhead(long loops)
532     {
533         return Measure(() =>
534         {
535             ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
536             ulong result = 0;
537             for (long i = 0; i < loops; i++)
538             {
539                 var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
540                 var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
541
542                 result += maxValue + source + target;
543             }
544             Global.Trash = result;
545         });
546     }
547     */
548
549     [Fact(Skip = "performance test")]
550     public static void GetSourceTest()
551     {
552         using (var scope = new TempLinksTestScope())
553         {
554             var links = scope.Links;
555             ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
                 Iterations);
556
557             ulong counter = 0;
558
```

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

                        var sw = Stopwatch.StartNew();

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

                        var elapsedTime = sw.Elapsed;

                        var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

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

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

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

                        long counter = 0;

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

                        var sw = Stopwatch.StartNew();

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

                        var elapsedTime = sw.Elapsed;

                        var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                        links.Delete(firstLink);

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

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

                        ulong counter = 0;

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

                        var sw = Stopwatch.StartNew();

                        for (ulong i = 0; i < Iterations; i++)
```

```csharp
634                    {
635                        counter += links.GetTarget(firstLink);
636                    }
637
638                    var elapsedTime = sw.Elapsed;
639
640                    var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
641
642                    links.Delete(firstLink);
643
644                    ConsoleHelpers.Debug(
645                        "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                            ↪  second), counter result: {3}",
646                        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
647                }
648            }
649
650        [Fact(Skip = "performance test")]
651        public static void TestGetTargetInParallel()
652        {
653            using (var scope = new TempLinksTestScope())
654            {
655                var links = scope.Links;
656                ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
                    ↪  parallel.", Iterations);
657
658                long counter = 0;
659
660                //var firstLink = links.First();
661                var firstLink = links.Create();
662
663                var sw = Stopwatch.StartNew();
664
665                Parallel.For(0, Iterations, x =>
666                {
667                    Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
668                    //Interlocked.Increment(ref counter);
669                });
670
671                var elapsedTime = sw.Elapsed;
672
673                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
674
675                links.Delete(firstLink);
676
677                ConsoleHelpers.Debug(
678                    "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
679                    Iterations, elapsedTime, (long)iterationsPerSecond, counter);
680            }
681        }
682
683        // TODO: Заполнить базу данных перед тестом
684        /*
685        [Fact]
686        public void TestRandomSearchFixed()
687        {
688            var tempFilename = Path.GetTempFileName();
689
690            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
     ↪  DefaultLinksSizeStep))
691            {
692                long iterations = 64 * 1024 * 1024 /
     ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
693
694                ulong counter = 0;
695                var maxLink = links.Total;
696
697                ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
698
699                var sw = Stopwatch.StartNew();
700
701                for (var i = iterations; i > 0; i--)
702                {
703                    var source =
     ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
704                    var target =
     ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
705
```

```
706                         counter += links.Search(source, target);
707                     }
708
709                     var elapsedTime = sw.Elapsed;
710
711                     var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
712
713                     ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
    ↪  Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
    ↪  counter);
714                 }
715
716             File.Delete(tempFilename);
717         }*/
718
719         [Fact(Skip = "useless: O(0), was dependent on creation tests")]
720         public static void TestRandomSearchAll()
721         {
722             using (var scope = new TempLinksTestScope())
723             {
724                 var links = scope.Links;
725                 ulong counter = 0;
726
727                 var maxLink = links.Count();
728
729                 var iterations = links.Count();
730
731                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
    ↪  links.Count());
732
733                 var sw = Stopwatch.StartNew();
734
735                 for (var i = iterations; i > 0; i--)
736                 {
737                     var linksAddressRange = new
    ↪  Range<ulong>(_constants.PossibleInnerReferencesRange.Minimum, maxLink);
738
739                     var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
740                     var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
741
742                     counter += links.SearchOrDefault(source, target);
743                 }
744
745                 var elapsedTime = sw.Elapsed;
746
747                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
748
749                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
    ↪  Iterations per second), c: {3}",
750                     iterations, elapsedTime, (long)iterationsPerSecond, counter);
751             }
752         }
753
754         [Fact(Skip = "useless: O(0), was dependent on creation tests")]
755         public static void TestEach()
756         {
757             using (var scope = new TempLinksTestScope())
758             {
759                 var links = scope.Links;
760
761                 var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
762
763                 ConsoleHelpers.Debug("Testing Each function.");
764
765                 var sw = Stopwatch.StartNew();
766
767                 links.Each(counter.IncrementAndReturnTrue);
768
769                 var elapsedTime = sw.Elapsed;
770
771                 var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
772
773                 ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
    ↪  links per second)",
774                     counter, elapsedTime, (long)linksPerSecond);
775             }
776         }
777
778         /*
779         [Fact]
```

```
780        public static void TestForeach()
781        {
782            var tempFilename = Path.GetTempFileName();
783
784            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪ DefaultLinksSizeStep))
785            {
786                ulong counter = 0;
787
788                ConsoleHelpers.Debug("Testing foreach through links.");
789
790                var sw = Stopwatch.StartNew();
791
792                //foreach (var link in links)
793                //{
794                //    counter++;
795                //}
796
797                var elapsedTime = sw.Elapsed;
798
799                var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
800
801                ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
    ↪ links per second)", counter, elapsedTime, (long)linksPerSecond);
802            }
803
804            File.Delete(tempFilename);
805        }
806        */
807
808        /*
809        [Fact]
810        public static void TestParallelForeach()
811        {
812            var tempFilename = Path.GetTempFileName();
813
814            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪ DefaultLinksSizeStep))
815            {
816
817                long counter = 0;
818
819                ConsoleHelpers.Debug("Testing parallel foreach through links.");
820
821                var sw = Stopwatch.StartNew();
822
823                //Parallel.ForEach((IEnumerable<ulong>)links, x =>
824                //{
825                //    Interlocked.Increment(ref counter);
826                //});
827
828                var elapsedTime = sw.Elapsed;
829
830                var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
831
832                ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
    ↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
833            }
834
835            File.Delete(tempFilename);
836        }
837        */
838
839        [Fact(Skip = "performance test")]
840        public static void Create64BillionLinks()
841        {
842            using (var scope = new TempLinksTestScope())
843            {
844                var links = scope.Links;
845                var linksBeforeTest = links.Count();
846
847                long linksToCreate = 64 * 1024 * 1024 /
                    ↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
848
849                ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
850
851                var elapsedTime = Performance.Measure(() =>
852                {
853                    for (long i = 0; i < linksToCreate; i++)
854                    {
```

```
855                            links.Create();
856                        }
857                    });

859                    var linksCreated = links.Count() - linksBeforeTest;
860                    var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;

862                    ConsoleHelpers.Debug("Current links count: {0}.", links.Count());

864                    ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
      ↪            linksCreated, elapsedTime,
865                        (long)linksPerSecond);
866                }
867            }

869            [Fact(Skip = "performance test")]
870            public static void Create64BillionLinksInParallel()
871            {
872                using (var scope = new TempLinksTestScope())
873                {
874                    var links = scope.Links;
875                    var linksBeforeTest = links.Count();

877                    var sw = Stopwatch.StartNew();

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

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

883                    Parallel.For(0, linksToCreate, x => links.Create());

885                    var elapsedTime = sw.Elapsed;

887                    var linksCreated = links.Count() - linksBeforeTest;
888                    var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;

890                    ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
      ↪            linksCreated, elapsedTime,
891                        (long)linksPerSecond);
892                }
893            }

895            [Fact(Skip = "useless: O(0), was dependent on creation tests")]
896            public static void TestDeletionOfAllLinks()
897            {
898                using (var scope = new TempLinksTestScope())
899                {
900                    var links = scope.Links;
901                    var linksBeforeTest = links.Count();

903                    ConsoleHelpers.Debug("Deleting all links");

905                    var elapsedTime = Performance.Measure(links.DeleteAll);

907                    var linksDeleted = linksBeforeTest - links.Count();
908                    var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;

910                    ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
      ↪            linksDeleted, elapsedTime,
911                        (long)linksPerSecond);
912                }
913            }

915            #endregion
916        }
917    }
```

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

```
1    using Xunit;
2    using Platform.Random;
3    using Platform.Data.Doublets.Numbers.Unary;

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
17  {
18      public static class UnicodeConvertersTests
19      {
20          [Fact]
21          public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
22          {
23              using (var scope = new TempLinksTestScope())
24              {
25                  var links = scope.Links;
26                  var meaningRoot = links.CreatePoint();
27                  var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
28                  var powerOf2ToUnaryNumberConverter = new
                    ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
29                  var addressToUnaryNumberConverter = new
                    ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
30                  var unaryNumberToAddressConverter = new
                    ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪ powerOf2ToUnaryNumberConverter);
31                  TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                    ↪ addressToUnaryNumberConverter, unaryNumberToAddressConverter);
32              }
33          }
34
35          [Fact]
36          public static void CharAndRawNumberUnicodeSymbolConvertersTest()
37          {
```

```csharp
38              using (var scope = new Scope<Types<HeapResizableDirectMemory,
    ↪   ResizableDirectMemoryLinks<ulong>>>())
39              {
40                  var links = scope.Use<ILinks<ulong>>();
41                  var meaningRoot = links.CreatePoint();
42                  var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
43                  var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
44                  TestCharAndUnicodeSymbolConverters(links, meaningRoot,
    ↪   addressToRawNumberConverter, rawNumberToAddressConverter);
45              }
46          }
47
48          private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
    ↪   meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
    ↪   numberToAddressConverter)
49          {
50              var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
51              var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
    ↪   addressToNumberConverter, unicodeSymbolMarker);
52              var originalCharacter = 'H';
53              var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
54              var unicodeSymbolCriterionMatcher = new UnicodeSymbolCriterionMatcher<ulong>(links,
    ↪   unicodeSymbolMarker);
55              var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
    ↪   numberToAddressConverter, unicodeSymbolCriterionMatcher);
56              var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
57              Assert.Equal(originalCharacter, resultingCharacter);
58          }
59
60          [Fact]
61          public static void StringAndUnicodeSequenceConvertersTest()
62          {
63              using (var scope = new TempLinksTestScope())
64              {
65                  var links = scope.Links;
66
67                  var itself = links.Constants.Itself;
68
69                  var meaningRoot = links.CreatePoint();
70                  var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
71                  var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
72                  var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
73                  var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
74                  var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
75
76                  var powerOf2ToUnaryNumberConverter = new
    ↪   PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
77                  var addressToUnaryNumberConverter = new
    ↪   AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
78                  var charToUnicodeSymbolConverter = new
    ↪   CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
    ↪   unicodeSymbolMarker);
79
80                  var unaryNumberToAddressConverter = new
    ↪   UnaryNumberToAddressOrOperationConverter<ulong>(links,
    ↪   powerOf2ToUnaryNumberConverter);
81                  var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
82                  var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↪   frequencyMarker, unaryOne, unaryNumberIncrementer);
83                  var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
    ↪   frequencyPropertyMarker, frequencyMarker);
84                  var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
    ↪   frequencyPropertyOperator, frequencyIncrementer);
85                  var linkToItsFrequencyNumberConverter = new
    ↪   LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
    ↪   unaryNumberToAddressConverter);
86                  var sequenceToItsLocalElementLevelsConverter = new
    ↪   SequenceToItsLocalElementLevelsConverter<ulong>(links,
    ↪   linkToItsFrequencyNumberConverter);
87                  var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↪   sequenceToItsLocalElementLevelsConverter);
88
89                  var stringToUnicodeSequenceConverter = new
    ↪   StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
    ↪   index, optimalVariantConverter, unicodeSequenceMarker);
90
91                  var originalString = "Hello";
```

```csharp
                var unicodeSequenceLink =
                    stringToUnicodeSequenceConverter.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