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

```
#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

```
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

```
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);
```

```

        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        ↪  => Links.Each(handler, restrictions);

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

        public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
        ↪  Links.Update(restrictions, substitution);

        public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);
    }
}
```

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

```
using System;
using System.Collections.Generic;
using Platform.Disposables;

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

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

        public ILinks<TLink> Links { get; }

        protected LinksDisposableDecoratorBase(ILinks<TLink> links)
        {
            Links = links;
            Constants = links.Constants;
        }

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

        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        ↪  => Links.Each(handler, restrictions);

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

        public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
        ↪  Links.Update(restrictions, substitution);

        public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);

        protected override bool AllowMultipleDisposeCalls => true;

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                Links.DisposeIfPossible();
            }
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
    ↪  be external (hybrid link's raw number).
    public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
    {
        public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }

        public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
            return Links.Each(handler, restrictions);
        }

        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
```

```
        {
            // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
            Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
            Links.EnsureInnerReferenceExists(substitution, nameof(substitution));
            return Links.Update(restrictions, substitution);
        }

        public override void Delete(IList<TLink> restrictions)
        {
            var link = restrictions[Constants.IndexPart];
            Links.EnsureLinkExists(link, nameof(link));
            Links.Delete(link);
        }
    }
}
```

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

```
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 class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

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

        public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            var constants = Constants;
            var itselfConstant = constants.Itself;
            var indexPartConstant = constants.IndexPart;
            var sourcePartConstant = constants.SourcePart;
            var targetPartConstant = constants.TargetPart;
            var restrictionsCount = restrictions.Count;
            if (!_equalityComparer.Equals(constants.Any, itselfConstant)
             && (((restrictionsCount > indexPartConstant) &&
              ↪  _equalityComparer.Equals(restrictions[indexPartConstant], itselfConstant))
             || ((restrictionsCount > sourcePartConstant) &&
              ↪  _equalityComparer.Equals(restrictions[sourcePartConstant], itselfConstant))
             || ((restrictionsCount > targetPartConstant) &&
              ↪  _equalityComparer.Equals(restrictions[targetPartConstant], itselfConstant))))
            {
                // Itself constant is not supported for Each method right now, skipping execution
                return constants.Continue;
            }
            return Links.Each(handler, restrictions);
        }

        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
        ↪  Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Itself,
        ↪  restrictions, substitution));
    }
}
```

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

```
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Decorators
{
    /// <remarks>
    /// Not practical if newSource and newTarget are too big.
    /// To be able to use practical version we should allow to create link at any specific
    ↪  location inside ResizableDirectMemoryLinks.
    /// This in turn will require to implement not a list of empty links, but a list of ranges
    ↪  to store it more efficiently.
    /// </remarks>
    public class LinksNonExistentDependenciesCreator<TLink> : LinksDecoratorBase<TLink>
    {
        public LinksNonExistentDependenciesCreator(ILinks<TLink> links) : base(links) { }

        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
```

```
            var constants = Constants;
            Links.EnsureCreated(substitution[constants.SourcePart],
            ↪  substitution[constants.TargetPart]);
            return Links.Update(restrictions, substitution);
        }
    }
}
```

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

```
using System.Collections.Generic;

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

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

        public override TLink Create(IList<TLink> restrictions)
        {
            var link = Links.Create();
            return Links.Update(link, link, link);
        }

        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
        ↪  Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Null,
        ↪  restrictions, substitution));
    }
}
```

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

```
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Decorators
{
    public class LinksUniquenessResolver<TLink> : LinksDecoratorBase<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

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

        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            var newLinkAddress = Links.SearchOrDefault(substitution[Constants.SourcePart],
            ↪  substitution[Constants.TargetPart]);
            if (_equalityComparer.Equals(newLinkAddress, default))
            {
                return Links.Update(restrictions, substitution);
            }
            return ResolveAddressChangeConflict(restrictions[Constants.IndexPart],
            ↪  newLinkAddress);
        }

        protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
        ↪  newLinkAddress)
        {
            if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
            ↪  Links.Exists(oldLinkAddress))
            {
                Facade.Delete(oldLinkAddress);
            }
            return newLinkAddress;
        }
    }
}
```

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

```
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Decorators
{
    public class LinksUniquenessValidator<TLink> : LinksDecoratorBase<TLink>
```

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

        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            Links.EnsureDoesNotExists(substitution[Constants.SourcePart],
            ↪  substitution[Constants.TargetPart]);
            return Links.Update(restrictions, substitution);
        }
    }
}
```

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

```
using System.Collections.Generic;

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

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

        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
            return Links.Update(restrictions, substitution);
        }

        public override void Delete(IList<TLink> restrictions)
        {
            var link = restrictions[Constants.IndexPart];
            Links.EnsureNoUsages(link);
            Links.Delete(link);
        }
    }
}
```

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

```
using System.Collections.Generic;

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

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

        public override void Delete(IList<TLink> restrictions)
        {
            var linkIndex = restrictions[Constants.IndexPart];
            Links.EnforceResetValues(linkIndex);
            Links.Delete(linkIndex);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// Представляет объект для работы с базой данных (файлом) в формате Links (массива связей).
    /// </summary>
    /// <remarks>
    /// Возможные оптимизации:
    /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
    ///     + меньше объём БД
    ///     - меньше производительность
    ///     - больше ограничение на количество связей в БД)
    /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
    ///     + меньше объём БД
    ///     - больше сложность
```

```
21      ///
22      /// 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
23      /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
        ↪  (битовыми строками) - вариант матрицы (выстраеваемой лениво).
24      ///
25      /// Решить отключать ли проверки при компиляции под Release. T.e. исключения будут
        ↪  выбрасываться только при #if DEBUG
26      /// </remarks>
27      public class UInt64Links : LinksDisposableDecoratorBase<ulong>
28      {
29          public UInt64Links(ILinks<ulong> links) : base(links) { }
30
31          public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
32          {
33              this.EnsureLinkIsAnyOrExists(restrictions);
34              return Links.Each(handler, restrictions);
35          }
36
37          public override ulong Create(IList<ulong> restrictions) => Links.CreatePoint();
38
39          public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
40          {
41              var constants = Constants;
42              var nullConstant = constants.Null;
43              if (restrictions.IsNullOrEmpty())
44              {
45                  return nullConstant;
46              }
47              // TODO: Looks like this is a common type of exceptions linked with restrictions
                ↪  support
48              if (substitution.Count != 3)
49              {
50                  throw new NotSupportedException();
51              }
52              var indexPartConstant = constants.IndexPart;
53              var updatedLink = restrictions[indexPartConstant];
54              this.EnsureLinkExists(updatedLink,
                ↪  $"{nameof(restrictions)}[{nameof(indexPartConstant)}]");
55              var sourcePartConstant = constants.SourcePart;
56              var newSource = substitution[sourcePartConstant];
57              this.EnsureLinkIsItselfOrExists(newSource,
                ↪  $"{nameof(substitution)}[{nameof(sourcePartConstant)}]");
58              var targetPartConstant = constants.TargetPart;
59              var newTarget = substitution[targetPartConstant];
60              this.EnsureLinkIsItselfOrExists(newTarget,
                ↪  $"{nameof(substitution)}[{nameof(targetPartConstant)}]");
61              var existedLink = nullConstant;
62              var itselfConstant = constants.Itself;
63              if (newSource != itselfConstant && newTarget != itselfConstant)
64              {
65                  existedLink = this.SearchOrDefault(newSource, newTarget);
66              }
67              if (existedLink == nullConstant)
68              {
69                  var before = Links.GetLink(updatedLink);
70                  if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
                    ↪  newTarget)
71                  {
72                      Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
                        ↪  newSource,
73                                               newTarget == itselfConstant ? updatedLink :
                                                 ↪  newTarget);
74                  }
75                  return updatedLink;
76              }
77              else
78              {
79                  return this.MergeAndDelete(updatedLink, existedLink);
80              }
81          }
82
83          public override void Delete(IList<ulong> restrictions)
84          {
85              var linkIndex = restrictions[Constants.IndexPart];
86              Links.EnsureLinkExists(linkIndex);
87              Links.EnforceResetValues(linkIndex);
```

```
            this.DeleteAllUsages(linkIndex);
            Links.Delete(linkIndex);
        }
    }
}
```

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

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

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

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

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

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

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

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

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

```
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
```

```
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
```

```
206             {
207                 var before = ArrayPool<TLink>.Empty;
208                 // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
                    ↪ (пройти мимо) или пустить (взять)?
209                 if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪ Constants.Break))
210                 {
211                     return Constants.Break;
212                 }
213                 var after = (IList<TLink>)substitution.ToArray();
214                 if (_equalityComparer.Equals(after[0], default))
215                 {
216                     var newLink = Links.Create();
217                     after[0] = newLink;
218                 }
219                 if (substitution.Count == 1)
220                 {
221                     after = Links.GetLink(substitution[0]);
222                 }
223                 else if (substitution.Count == 3)
224                 {
225                     //Links.Create(after);
226                 }
227                 else
228                 {
229                     throw new NotSupportedException();
230                 }
231                 if (matchHandler != null)
232                 {
233                     return substitutionHandler(before, after);
234                 }
235                 return Constants.Continue;
236             }
237             else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
238             {
239                 if (patternOrCondition.Count == 1)
240                 {
241                     var linkToDelete = patternOrCondition[0];
242                     var before = Links.GetLink(linkToDelete);
243                     if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                        ↪ Constants.Break))
244                     {
245                         return Constants.Break;
246                     }
247                     var after = ArrayPool<TLink>.Empty;
248                     Links.Update(linkToDelete, Constants.Null, Constants.Null);
249                     Links.Delete(linkToDelete);
250                     if (matchHandler != null)
251                     {
252                         return substitutionHandler(before, after);
253                     }
254                     return Constants.Continue;
255                 }
256                 else
257                 {
258                     throw new NotSupportedException();
259                 }
260             }
261             else // Replace / Update
262             {
263                 if (patternOrCondition.Count == 1) //-V3125
264                 {
265                     var linkToUpdate = patternOrCondition[0];
266                     var before = Links.GetLink(linkToUpdate);
267                     if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                        ↪ Constants.Break))
268                     {
269                         return Constants.Break;
270                     }
271                     var after = (IList<TLink>)substitution.ToArray(); //-V3125
272                     if (_equalityComparer.Equals(after[0], default))
273                     {
274                         after[0] = linkToUpdate;
275                     }
276                     if (substitution.Count == 1)
277                     {
278                         if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
279                         {
```

```
                            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

```
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

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

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

namespace Platform.Data.Doublets
{
    public struct Doublet<T> : IEquatable<Doublet<T>>
    {
        private static readonly EqualityComparer<T> _equalityComparer =
        ↪  EqualityComparer<T>.Default;

        public T Source { get; set; }
        public T Target { get; set; }

        public Doublet(T source, T target)
        {
            Source = source;
            Target = target;
        }

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

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

        public override bool Equals(object obj) => obj is Doublet<T> doublet ?
        ↪  base.Equals(doublet) : false;

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

./Platform.Data.Doublets/Hybrid.cs

```
using System;
using System.Reflection;
using System.Reflection.Emit;
using Platform.Reflection;
using Platform.Converters;
using Platform.Exceptions;

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

namespace Platform.Data.Doublets
{
    public class Hybrid<T>
    {
        private static readonly Func<object, T> _absAndConvert;
        private static readonly Func<object, T> _absAndNegateAndConvert;

        static Hybrid()
        {
            _absAndConvert = DelegateHelpers.Compile<Func<object, T>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<T>();
                emiter.LoadArgument(0);
                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 unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                ↪  signedVersion });
                emiter.Call(unsignedMethod);
                emiter.Return();
            });
            _absAndNegateAndConvert = DelegateHelpers.Compile<Func<object, T>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<T>();
                emiter.LoadArgument(0);
```

```
            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);
```

```
107
108         public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;
109
110         public static explicit operator ulong(Hybrid<T> hybrid) =>
            ↪ Convert.ToUInt64(hybrid.Value);
111
112         public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;
113
114         public static explicit operator uint(Hybrid<T> hybrid) => Convert.ToUInt32(hybrid.Value);
115
116         public static explicit operator int(Hybrid<T> hybrid) =>
            ↪ Convert.ToInt32(hybrid.AbsoluteValue);
117
118         public static explicit operator ushort(Hybrid<T> hybrid) =>
            ↪ Convert.ToUInt16(hybrid.Value);
119
120         public static explicit operator short(Hybrid<T> hybrid) =>
            ↪ Convert.ToInt16(hybrid.AbsoluteValue);
121
122         public static explicit operator byte(Hybrid<T> hybrid) => Convert.ToByte(hybrid.Value);
123
124         public static explicit operator sbyte(Hybrid<T> hybrid) =>
            ↪ Convert.ToSByte(hybrid.AbsoluteValue);
125
126         public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
            ↪ default(T).ToString() : IsExternal ? $"<{AbsoluteValue}>" : Value.ToString();
127     }
128 }
```

./Platform.Data.Doublets/ILinks.cs

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

./Platform.Data.Doublets/ILinksExtensions.cs

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Runtime.CompilerServices;
6  using Platform.Ranges;
7  using Platform.Collections.Arrays;
8  using Platform.Numbers;
9  using Platform.Random;
10 using Platform.Setters;
11 using Platform.Data.Exceptions;
12 using Platform.Data.Doublets.Decorators;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets
17 {
18     public static class ILinksExtensions
19     {
20         public static void RunRandomCreations<TLink>(this ILinks<TLink> links, long
            ↪ amountOfCreations)
21         {
22             for (long i = 0; i < amountOfCreations; i++)
23             {
24                 var linksAddressRange = new Range<ulong>(0, (Integer<TLink>)links.Count());
25                 Integer<TLink> source = RandomHelpers.Default.NextUInt64(linksAddressRange);
26                 Integer<TLink> target = RandomHelpers.Default.NextUInt64(linksAddressRange);
27                 links.CreateAndUpdate(source, target);
28             }
29         }
30
31         public static void RunRandomSearches<TLink>(this ILinks<TLink> links, long
            ↪ amountOfSearches)
32         {
33             for (long i = 0; i < amountOfSearches; i++)
34             {
35                 var linkAddressRange = new Range<ulong>(1, (Integer<TLink>)links.Count());
```

```
36                  Integer<TLink> source = RandomHelpers.Default.NextUInt64(linkAddressRange);
37                  Integer<TLink> target = RandomHelpers.Default.NextUInt64(linkAddressRange);
38                  links.SearchOrDefault(source, target);
39              }
40          }
41
42          public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, long
            ↪  amountOfDeletions)
43          {
44              var min = (ulong)amountOfDeletions > (Integer<TLink>)links.Count() ? 1 :
                ↪  (Integer<TLink>)links.Count() - (ulong)amountOfDeletions;
45              for (long i = 0; i < amountOfDeletions; i++)
46              {
47                  var linksAddressRange = new Range<ulong>(min, (Integer<TLink>)links.Count());
48                  Integer<TLink> link = RandomHelpers.Default.NextUInt64(linksAddressRange);
49                  links.Delete(link);
50                  if ((Integer<TLink>)links.Count() < min)
51                  {
52                      break;
53                  }
54              }
55          }
56
57          public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
            ↪  links.Delete(new LinkAddress<TLink>(linkToDelete));
58
59          /// <remarks>
60          /// TODO: Возможно есть очень простой способ это сделать.
61          /// (Например просто удалить файл, или изменить его размер таким образом,
62          /// чтобы удалился весь контент)
63          /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
64          /// </remarks>
65          public static void DeleteAll<TLink>(this ILinks<TLink> links)
66          {
67              var equalityComparer = EqualityComparer<TLink>.Default;
68              var comparer = Comparer<TLink>.Default;
69              for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
                ↪  Arithmetic.Decrement(i))
70              {
71                  links.Delete(i);
72                  if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
73                  {
74                      i = links.Count();
75                  }
76              }
77          }
78
79          public static TLink First<TLink>(this ILinks<TLink> links)
80          {
81              TLink firstLink = default;
82              var equalityComparer = EqualityComparer<TLink>.Default;
83              if (equalityComparer.Equals(links.Count(), default))
84              {
85                  throw new InvalidOperationException("В хранилище нет связей.");
86              }
87              links.Each(links.Constants.Any, links.Constants.Any, link =>
88              {
89                  firstLink = link[links.Constants.IndexPart];
90                  return links.Constants.Break;
91              });
92              if (equalityComparer.Equals(firstLink, default))
93              {
94                  throw new InvalidOperationException("В процессе поиска по хранилищу не было
                    ↪  найдено связей.");
95              }
96              return firstLink;
97          }
98
99          #region Paths
100
101         /// <remarks>
102         /// TODO: Как так? Как то что ниже может быть корректно?
103         /// Скорее всего практически не применимо
104         /// Предполагалось, что можно было конвертировать формируемый в проходе через
            ↪  SequenceWalker
105         /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
106         /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
107         /// </remarks>
```

```
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.
```

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

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

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

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

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

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

        /// <summary>
        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
        ↪  (handler) для каждой подходящей связи.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Значение, определяющее соответствующие шаблону связи.
        ↪  (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
        ↪  Constants.Any - любое начало, 1..∞ конкретное начало)</param>
```

```
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)]
```

```
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         }
```

```
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             {
```

```
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>
```

```
565             /// <param name="links">Хранилище связей.</param>
566             /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
567             /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
568             [MethodImpl(MethodImplOptions.AggressiveInlining)]
569             public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
                ↪  target)
570             {
571                 var link = links.SearchOrDefault(source, target);
572                 if (!EqualityComparer<TLink>.Default.Equals(link, default))
573                 {
574                     links.Delete(link);
575                     return link;
576                 }
577                 return default;
578             }
579
580             /// <summary>Удаляет несколько связей.</summary>
581             /// <param name="links">Хранилище связей.</param>
582             /// <param name="deletedLinks">Список адресов связей к удалению.</param>
583             [MethodImpl(MethodImplOptions.AggressiveInlining)]
584             public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
585             {
586                 for (int i = 0; i < deletedLinks.Count; i++)
587                 {
588                     links.Delete(deletedLinks[i]);
589                 }
590             }
591
592             /// <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>
593             public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
594             {
595                 var anyConstant = links.Constants.Any;
596                 var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
597                 links.DeleteByQuery(usagesAsSourceQuery);
598                 var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
599                 links.DeleteByQuery(usagesAsTargetQuery);
600             }
601
602             public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
603             {
604                 var count = (Integer<TLink>)links.Count(query);
605                 if (count > 0)
606                 {
607                     var queryResult = new TLink[count];
608                     var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
                        ↪  links.Constants.Continue);
609                     links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
610                     for (var i = (long)count - 1; i >= 0; i--)
611                     {
612                         links.Delete(queryResult[i]);
613                     }
614                 }
615             }
616
617             // TODO: Move to Platform.Data
618             public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
619             {
620                 var nullConstant = links.Constants.Null;
621                 var equalityComparer = EqualityComparer<TLink>.Default;
622                 var link = links.GetLink(linkIndex);
623                 for (int i = 1; i < link.Count; i++)
624                 {
625                     if (!equalityComparer.Equals(link[i], nullConstant))
626                     {
627                         return false;
628                     }
629                 }
630                 return true;
631             }
632
633             // TODO: Create a universal version of this method in Platform.Data (with using of for
                ↪  loop)
634             public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
635             {
636                 var nullConstant = links.Constants.Null;
637                 var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
```

```
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).
```

```
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
        ↪  TLink newLinkIndex)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
            {
                links.MergeUsages(oldLinkIndex, newLinkIndex);
                links.Delete(oldLinkIndex);
            }
            return newLinkIndex;
        }

        public static ILinks<TLink>
        ↪  DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
        {
            links = new LinksCascadeUsagesResolver<TLink>(links);
            links = new NonNullContentsLinkDeletionResolver<TLink>(links);
            links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
            return links;
        }
    }
}
```

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

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

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

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

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

        public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
        ↪  IIncrementer<TLink> unaryNumberIncrementer)
            : base(links)
        {
            _frequencyMarker = frequencyMarker;
            _unaryOne = unaryOne;
            _unaryNumberIncrementer = unaryNumberIncrementer;
        }

        public TLink Increment(TLink frequency)
        {
            if (_equalityComparer.Equals(frequency, default))
            {
                return Links.GetOrCreate(_unaryOne, _frequencyMarker);
            }
            var source = Links.GetSource(frequency);
            var incrementedSource = _unaryNumberIncrementer.Increment(source);
            return Links.GetOrCreate(incrementedSource, _frequencyMarker);
        }
    }
}
```

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

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

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

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

        private readonly TLink _unaryOne;

```

```
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;
```

```
            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();
110
111         public IEnumerator<TLink> GetEnumerator()
112         {
113             yield return Index;
114             yield return Source;
115             yield return Target;
116         }
117
118         public void Add(TLink item) => throw new NotSupportedException();
119
120         public void Clear() => throw new NotSupportedException();
121
122         public bool Contains(TLink item) => IndexOf(item) >= 0;
123
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         }
136
137         public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
138
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         }
155
156         public void Insert(int index, TLink item) => throw new NotSupportedException();
157
158         public void RemoveAt(int index) => throw new NotSupportedException();
159
160         #endregion
161     }
162 }
```

./Platform.Data.Doublets/LinkExtensions.cs

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

./Platform.Data.Doublets/LinksOperatorBase.cs

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

```
    }
}
```

./Platform.Data.Doublets/Numbers/Raw/AddressToRawNumberConverter.cs
```
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Numbers.Raw
{
    public class AddressToRawNumberConverter<TLink> : IConverter<TLink>
    {
        public TLink Convert(TLink source) => new Hybrid<TLink>(source, isExternal: true);
    }
}
```

./Platform.Data.Doublets/Numbers/Raw/RawNumberToAddressConverter.cs
```
using Platform.Interfaces;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Numbers.Raw
{
    public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
    {
        public TLink Convert(TLink source) => (Integer<TLink>)new
        ↪ Hybrid<TLink>(source).AbsoluteValue;
    }
}
```

./Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs
```
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Reflection;
using Platform.Numbers;

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

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

        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;

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

        public TLink Convert(TLink number)
        {
            var nullConstant = Links.Constants.Null;
            var one = Integer<TLink>.One;
            var target = nullConstant;
            for (int i = 0; !_equalityComparer.Equals(number, default) && i <
            ↪ NumericType<TLink>.BitsLength; i++)
            {
                if (_equalityComparer.Equals(Bit.And(number, one), one))
                {
                    target = _equalityComparer.Equals(target, nullConstant)
                        ? _powerOf2ToUnaryNumberConverter.Convert(i)
                        : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
                }
                number = Bit.ShiftRight(number, 1);
            }
            return target;
        }
    }
}
```

./Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConveter.cs
```
using System;
using System.Collections.Generic;
using Platform.Interfaces;

```

```
#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

```
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;
        }
    }
}
```

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

```
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);
    }
}
```

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

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

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

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

        private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;

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

        public TLink Convert(TLink sourceNumber)
        {
            var nullConstant = Links.Constants.Null;
            var source = sourceNumber;
            var target = nullConstant;
            if (!_equalityComparer.Equals(source, nullConstant))
            {
                while (true)
                {
                    if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
                    {
                        SetBit(ref target, powerOf2Index);
                        break;
                    }
                    else
                    {
                        powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
                        SetBit(ref target, powerOf2Index);
                        source = Links.GetTarget(source);
                    }
                }
            }
            return target;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void SetBit(ref TLink target, int powerOf2Index) => target =
        ↪ Bit.Or(target, Bit.ShiftLeft(Integer<TLink>.One, powerOf2Index));
    }
}
```

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

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

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

namespace Platform.Data.Doublets.PropertyOperators
{
    public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>,
    ↪ IPropertiesOperator<TLink, TLink, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;

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

        public TLink GetValue(TLink @object, TLink property)
        {
```

```
            var objectProperty = Links.SearchOrDefault(@object, property);
            if (_equalityComparer.Equals(objectProperty, default))
            {
                return default;
            }
            var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
            if (valueLink == null)
            {
                return default;
            }
            return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
        }

        public void SetValue(TLink @object, TLink property, TLink value)
        {
            var objectProperty = Links.GetOrCreate(@object, property);
            Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
            Links.GetOrCreate(objectProperty, value);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.PropertyOperators
{
    public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IPropertyOperator<TLink,
    ↪ TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;

        private readonly TLink _propertyMarker;
        private readonly TLink _propertyValueMarker;

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

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

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

```

```
55          private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
            ↪  ? default : Links.GetTarget(container);
56
57          public void Set(TLink link, TLink value)
58          {
59              var property = Links.GetOrCreate(link, _propertyMarker);
60              var container = GetContainer(property);
61              if (_equalityComparer.Equals(container, default))
62              {
63                  Links.GetOrCreate(property, value);
64              }
65              else
66              {
67                  Links.Update(container, property, value);
68              }
69          }
70      }
71  }
```

./Platform.Data.Doublets/ResizableDirectMemory/ILinksListMethods.cs

```
1   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3   namespace Platform.Data.Doublets.ResizableDirectMemory
4   {
5       public interface ILinksListMethods<TLink>
6       {
7           void Detach(TLink freeLink);
8           void AttachAsFirst(TLink link);
9       }
10  }
```

./Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.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.ResizableDirectMemory
7   {
8       public interface ILinksTreeMethods<TLink>
9       {
10          TLink CountUsages(TLink link);
11          TLink Search(TLink source, TLink target);
12          TLink EachUsage(TLink source, Func<IList<TLink>, TLink> handler);
13          void Detach(ref TLink firstAsSource, TLink linkIndex);
14          void Attach(ref TLink firstAsSource, TLink linkIndex);
15      }
16  }
```

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

```
1   using System;
2   using System.Text;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Numbers;
6   using Platform.Collections.Methods.Trees;
7   using static System.Runtime.CompilerServices.Unsafe;
8
9   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11  namespace Platform.Data.Doublets.ResizableDirectMemory
12  {
13      public unsafe abstract class LinksAVLBalancedTreeMethodsBase<TLink> :
        ↪  SizedAndThreadedAVLBalancedTreeMethods<TLink>
14      {
15          private readonly ResizableDirectMemoryLinks<TLink> _memory;
16          private readonly LinksConstants<TLink> _constants;
17          protected readonly byte* Links;
18          protected readonly byte* Header;
19
20          public LinksAVLBalancedTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory, byte*
            ↪  links, byte* header)
21          {
22              Links = links;
23              Header = header;
24              _memory = memory;
25              _constants = memory.Constants;
26          }
27
```

```
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected abstract TLink GetTreeRoot();
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected abstract TLink GetBasePartValue(TLink link);
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
            ↪  rootSource, TLink rootTarget);
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
            ↪  rootSource, TLink rootTarget);
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          internal virtual ref LinksHeader<TLink> GetHeaderReference() => ref
            ↪  AsRef<LinksHeader<TLink>>(Header);
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          internal virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
            ↪  AsRef<RawLink<TLink>>((void*)(Links + RawLink<TLink>.SizeInBytes *
            ↪  (Integer<TLink>)link));
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          protected virtual Link<TLink> GetLinkValues(TLink current) =>
            ↪  _memory.GetLinkStruct(current);
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
51          {
52              ref var firstLink = ref GetLinkReference(first);
53              ref var secondLink = ref GetLinkReference(second);
54              return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
55          }
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
59          {
60              ref var firstLink = ref GetLinkReference(first);
61              ref var secondLink = ref GetLinkReference(second);
62              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
63          }
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          protected static TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5, -5);
67
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          protected static void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
            ↪  Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
70
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          protected bool GetLeftIsChildValue(TLink value)
73          {
74              unchecked
75              {
76                  //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
77                  return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
78              }
79          }
80
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          protected static void SetLeftIsChildValue(ref TLink storedValue, bool value)
83          {
84              unchecked
85              {
86                  var previousValue = storedValue;
87                  var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
                    ↪  1);
88                  storedValue = modified;
89              }
90          }
91
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          protected bool GetRightIsChildValue(TLink value)
94          {
95              unchecked
```

```
96              {
97                  //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
98                  return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
99              }
100         }
101
102         [MethodImpl(MethodImplOptions.AggressiveInlining)]
103         protected static void SetRightIsChildValue(ref TLink storedValue, bool value)
104         {
105             unchecked
106             {
107                 var previousValue = storedValue;
108                 var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
                        ↪  1);
109                 storedValue = modified;
110             }
111         }
112
113         [MethodImpl(MethodImplOptions.AggressiveInlining)]
114         protected static sbyte GetBalanceValue(TLink storedValue)
115         {
116             unchecked
117             {
118                 var value = (int)(Integer<TLink>)Bit<TLink>.PartialRead(storedValue, 0, 3);
119                 value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
                        ↪  end of sbyte
120                 return (sbyte)value;
121             }
122         }
123
124         [MethodImpl(MethodImplOptions.AggressiveInlining)]
125         protected static void SetBalanceValue(ref TLink storedValue, sbyte value)
126         {
127             unchecked
128             {
129                 var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
                        ↪  3);
130                 var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
131                 storedValue = modified;
132             }
133         }
134
135         public TLink this[TLink index]
136         {
137             get
138             {
139                 var root = GetTreeRoot();
140                 if (GreaterOrEqualThan(index, GetSize(root)))
141                 {
142                     return Zero;
143                 }
144                 while (!EqualToZero(root))
145                 {
146                     var left = GetLeftOrDefault(root);
147                     var leftSize = GetSizeOrZero(left);
148                     if (LessThan(index, leftSize))
149                     {
150                         root = left;
151                         continue;
152                     }
153                     if (IsEquals(index, leftSize))
154                     {
155                         return root;
156                     }
157                     root = GetRightOrDefault(root);
158                     index = Subtract(index, Increment(leftSize));
159                 }
160                 return Zero; // TODO: Impossible situation exception (only if tree structure
                        ↪  broken)
161             }
162         }
163
164         /// <summary>
165         /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪  (концом).
166         /// </summary>
167         /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
168         /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
169         /// <returns>Индекс искомой связи.</returns>
```

```
170         public TLink Search(TLink source, TLink target)
171         {
172             var root = GetTreeRoot();
173             while (!EqualToZero(root))
174             {
175                 ref var rootLink = ref GetLinkReference(root);
176                 var rootSource = rootLink.Source;
177                 var rootTarget = rootLink.Target;
178                 if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                    ↪ node.Key < root.Key
179                 {
180                     root = GetLeftOrDefault(root);
181                 }
182                 else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                    ↪ node.Key > root.Key
183                 {
184                     root = GetRightOrDefault(root);
185                 }
186                 else // node.Key == root.Key
187                 {
188                     return root;
189                 }
190             }
191             return Zero;
192         }
193 
194         // TODO: Return indices range instead of references count
195         public TLink CountUsages(TLink link)
196         {
197             var root = GetTreeRoot();
198             var total = GetSize(root);
199             var totalRightIgnore = Zero;
200             while (!EqualToZero(root))
201             {
202                 var @base = GetBasePartValue(root);
203                 if (LessOrEqualThan(@base, link))
204                 {
205                     root = GetRightOrDefault(root);
206                 }
207                 else
208                 {
209                     totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
210                     root = GetLeftOrDefault(root);
211                 }
212             }
213             root = GetTreeRoot();
214             var totalLeftIgnore = Zero;
215             while (!EqualToZero(root))
216             {
217                 var @base = GetBasePartValue(root);
218                 if (GreaterOrEqualThan(@base, link))
219                 {
220                     root = GetLeftOrDefault(root);
221                 }
222                 else
223                 {
224                     totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
225 
226                     root = GetRightOrDefault(root);
227                 }
228             }
229             return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
230         }
231 
232         public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
233         {
234             var root = GetTreeRoot();
235             if (EqualToZero(root))
236             {
237                 return _constants.Continue;
238             }
239             TLink first = Zero, current = root;
240             while (!EqualToZero(current))
241             {
242                 var @base = GetBasePartValue(current);
243                 if (GreaterOrEqualThan(@base, link))
244                 {
245                     if (IsEquals(@base, link))
246                     {
```

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

        protected override void PrintNodeValue(TLink node, StringBuilder sb)
        {
            ref var link = ref GetLinkReference(node);
            sb.Append(' ');
            sb.Append(link.Source);
            sb.Append('-');
            sb.Append('>');
            sb.Append(link.Target);
        }
    }
}
```

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

```
using Platform.Unsafe;

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    internal struct LinksHeader<TLink>
    {
        public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;

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

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

```
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 LinksSourcesAVLBalancedTreeMethods<TLink> :
    ↪ LinksAVLBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
    {
        public LinksSourcesAVLBalancedTreeMethods(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).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
17          protected unsafe override ref TLink GetRightReference(TLink node) => ref
            ↪ GetLinkReference(node).RightAsSource;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected override void SetLeft(TLink node, TLink left) =>
            ↪ GetLinkReference(node).LeftAsSource = left;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected override void SetRight(TLink node, TLink right) =>
            ↪ GetLinkReference(node).RightAsSource = right;
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected override TLink GetSize(TLink node) =>
            ↪ GetSizeValue(GetLinkReference(node).SizeAsSource);
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
            ↪ GetLinkReference(node).SizeAsSource, size);
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected override bool GetLeftIsChild(TLink node) =>
            ↪ GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected override void SetLeftIsChild(TLink node, bool value) =>
            ↪ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          protected override bool GetRightIsChild(TLink node) =>
            ↪ GetRightIsChildValue(GetLinkReference(node).SizeAsSource);
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          protected override void SetRightIsChild(TLink node, bool value) =>
            ↪ SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          protected override sbyte GetBalance(TLink node) =>
            ↪ GetBalanceValue(GetLinkReference(node).SizeAsSource);
51
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
53          protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
            ↪ GetLinkReference(node).SizeAsSource, value);
54
55          [MethodImpl(MethodImplOptions.AggressiveInlining)]
56          protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsSource;
57
58          [MethodImpl(MethodImplOptions.AggressiveInlining)]
59          protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
60
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
            ↪ (IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
63
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
            ↪ (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
66
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          protected override void ClearNode(TLink node)
69          {
70              ref var link = ref GetLinkReference(node);
71              link.LeftAsSource = Zero;
72              link.RightAsSource = Zero;
73              link.SizeAsSource = Zero;
74          }
75      }
76  }
```

./Platform.Data.Doublets/ResizableDirectMemory/LinksTargetsAVLBalancedTreeMethods.cs

```
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)]
```

```
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

```
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

```
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 байтах.</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;
```

```
40          private byte* _header;
41          private byte* _links;
42
43          private ILinksTreeMethods<TLink> _targetsTreeMethods;
44          private ILinksTreeMethods<TLink> _sourcesTreeMethods;
45
46          // 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
47          private ILinksListMethods<TLink> _unusedLinksListMethods;
48
49          /// <summary>
50          /// Возвращает общее число связей находящихся в хранилище.
51          /// </summary>
52          private TLink Total
53          {
54              get
55              {
56                  ref var header = ref AsRef<LinksHeader<TLink>>(_header);
57                  return Subtract(header.AllocatedLinks, header.FreeLinks);
58              }
59          }
60
61          public LinksConstants<TLink> Constants { get; }
62
63          public ResizableDirectMemoryLinks(string address) : this(address, DefaultLinksSizeStep)
            ↪  { }
64
65          /// <summary>
66          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪  минимальным шагом расширения базы данных.
67          /// </summary>
68          /// <param name="address">Полный пусть к файлу базы данных.</param>
69          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪  байтах.</param>
70          public ResizableDirectMemoryLinks(string address, long memoryReservationStep) : this(new
            ↪  FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪  memoryReservationStep) { }
71
72          public ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪  DefaultLinksSizeStep) { }
73
74          public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep)
75          {
76              Constants = Default<LinksConstants<TLink>>.Instance;
77              _memory = memory;
78              _memoryReservationStep = memoryReservationStep;
79              if (memory.ReservedCapacity < memoryReservationStep)
80              {
81                  memory.ReservedCapacity = memoryReservationStep;
82              }
83              SetPointers(_memory);
84              ref var header = ref AsRef<LinksHeader<TLink>>(_header);
85              // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
86              _memory.UsedCapacity = ((Integer<TLink>)header.AllocatedLinks * LinkSizeInBytes) +
                ↪  LinkHeaderSizeInBytes;
87              // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
88              header.ReservedLinks = (Integer<TLink>)((_memory.ReservedCapacity -
                ↪  LinkHeaderSizeInBytes) / LinkSizeInBytes);
89          }
90
91          [MethodImpl(MethodImplOptions.AggressiveInlining)]
92          public TLink Count(IList<TLink> restrictions)
93          {
94              // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
95              if (restrictions.Count == 0)
96              {
97                  return Total;
98              }
99              if (restrictions.Count == 1)
100             {
101                 var index = restrictions[Constants.IndexPart];
102                 if (_equalityComparer.Equals(index, Constants.Any))
103                 {
104                     return Total;
105                 }
106                 return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
107             }
```

```
108             if (restrictions.Count == 2)
109             {
110                 var index = restrictions[Constants.IndexPart];
111                 var value = restrictions[1];
112                 if (_equalityComparer.Equals(index, Constants.Any))
113                 {
114                     if (_equalityComparer.Equals(value, Constants.Any))
115                     {
116                         return Total; // Any - как отсутствие ограничения
117                     }
118                     return Add(_sourcesTreeMethods.CountUsages(value),
                        ↪ _targetsTreeMethods.CountUsages(value));
119                 }
120                 else
121                 {
122                     if (!Exists(index))
123                     {
124                         return Integer<TLink>.Zero;
125                     }
126                     if (_equalityComparer.Equals(value, Constants.Any))
127                     {
128                         return Integer<TLink>.One;
129                     }
130                     ref var storedLinkValue = ref GetLinkUnsafe(index);
131                     if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
132                         _equalityComparer.Equals(storedLinkValue.Target, value))
133                     {
134                         return Integer<TLink>.One;
135                     }
136                     return Integer<TLink>.Zero;
137                 }
138             }
139             if (restrictions.Count == 3)
140             {
141                 var index = restrictions[Constants.IndexPart];
142                 var source = restrictions[Constants.SourcePart];
143                 var target = restrictions[Constants.TargetPart];
144 
145                 if (_equalityComparer.Equals(index, Constants.Any))
146                 {
147                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ _equalityComparer.Equals(target, Constants.Any))
148                     {
149                         return Total;
150                     }
151                     else if (_equalityComparer.Equals(source, Constants.Any))
152                     {
153                         return _targetsTreeMethods.CountUsages(target);
154                     }
155                     else if (_equalityComparer.Equals(target, Constants.Any))
156                     {
157                         return _sourcesTreeMethods.CountUsages(source);
158                     }
159                     else //if(source != Any && target != Any)
160                     {
161                         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
162                         var link = _sourcesTreeMethods.Search(source, target);
163                         return _equalityComparer.Equals(link, Constants.Null) ?
                            ↪ Integer<TLink>.Zero : Integer<TLink>.One;
164                     }
165                 }
166                 else
167                 {
168                     if (!Exists(index))
169                     {
170                         return Integer<TLink>.Zero;
171                     }
172                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ _equalityComparer.Equals(target, Constants.Any))
173                     {
174                         return Integer<TLink>.One;
175                     }
176                     ref var storedLinkValue = ref GetLinkUnsafe(index);
177                     if (!_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ !_equalityComparer.Equals(target, Constants.Any))
178                     {
179                         if (_equalityComparer.Equals(storedLinkValue.Source, source) &&
180                             _equalityComparer.Equals(storedLinkValue.Target, target))
```

```
                    {
                        return Integer<TLink>.One;
                    }
                    return Integer<TLink>.Zero;
                }
                var value = default(TLink);
                if (_equalityComparer.Equals(source, Constants.Any))
                {
                    value = target;
                }
                if (_equalityComparer.Equals(target, Constants.Any))
                {
                    value = source;
                }
                if (_equalityComparer.Equals(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;
```

```
328                 }
329             }
330             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪ поддерживаются.");
331         }
332 
333         /// <remarks>
334         /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
                ↪ в другом месте (но не в менеджере памяти, а в логике Links)
335         /// </remarks>
336         [MethodImpl(MethodImplOptions.AggressiveInlining)]
337         public TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
338         {
339             var linkIndex = restrictions[Constants.IndexPart];
340             ref var link = ref GetLinkUnsafe(linkIndex);
341             ref var firstAsSource = ref AsRef<LinksHeader<TLink>>(_header).FirstAsSource;
342             ref var firstAsTarget = ref AsRef<LinksHeader<TLink>>(_header).FirstAsTarget;
343             // Будет корректно работать только в том случае, если пространство выделенной связи
                ↪ предварительно заполнено нулями
344             if (!_equalityComparer.Equals(link.Source, Constants.Null))
345             {
346                 _sourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
347             }
348             if (!_equalityComparer.Equals(link.Target, Constants.Null))
349             {
350                 _targetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
351             }
352             link.Source = substitution[Constants.SourcePart];
353             link.Target = substitution[Constants.TargetPart];
354             if (!_equalityComparer.Equals(link.Source, Constants.Null))
355             {
356                 _sourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
357             }
358             if (!_equalityComparer.Equals(link.Target, Constants.Null))
359             {
360                 _targetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
361             }
362             return linkIndex;
363         }
364 
365         [MethodImpl(MethodImplOptions.AggressiveInlining)]
366         public Link<TLink> GetLinkStruct(TLink linkIndex)
367         {
368             ref var link = ref GetLinkUnsafe(linkIndex);
369             return new Link<TLink>(linkIndex, link.Source, link.Target);
370         }
371 
372         [MethodImpl(MethodImplOptions.AggressiveInlining)]
373         internal ref RawLink<TLink> GetLinkUnsafe(TLink linkIndex) => ref
            ↪ AsRef<RawLink<TLink>>(_links + LinkSizeInBytes * (Integer<TLink>)linkIndex);
374 
375         /// <remarks>
376         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
                ↪ пространство
377         /// </remarks>
378         public TLink Create(IList<TLink> restrictions)
379         {
380             ref var header = ref AsRef<LinksHeader<TLink>>(_header);
381             var freeLink = header.FirstFreeLink;
382             if (!_equalityComparer.Equals(freeLink, Constants.Null))
383             {
384                 _unusedLinksListMethods.Detach(freeLink);
385             }
386             else
387             {
388                 var maximumPossibleInnerReference =
                    ↪ Constants.PossibleInnerReferencesRange.Maximum;
389                 if (_comparer.Compare(header.AllocatedLinks, maximumPossibleInnerReference) > 0)
390                 {
391                     throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
392                 }
393                 if (_comparer.Compare(header.AllocatedLinks, Decrement(header.ReservedLinks)) >=
                    ↪ 0)
394                 {
395                     _memory.ReservedCapacity += _memoryReservationStep;
396                     SetPointers(_memory);
397                     header.ReservedLinks = (Integer<TLink>)(_memory.ReservedCapacity /
                        ↪ LinkSizeInBytes);
```

```
            }
            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;

```

```
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                SetPointers(null);
                _memory.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksAVLBalancedTreeMethodsBase.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Text;
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 UInt64LinksAVLBalancedTreeMethodsBase :
    ↪ SizedAndThreadedAVLBalancedTreeMethods<ulong>
    {
        private readonly UInt64ResizableDirectMemoryLinks _memory;
        private readonly LinksConstants<ulong> _constants;
        internal readonly UInt64RawLink* _links;
        internal readonly UInt64LinksHeader* _header;

        internal UInt64LinksAVLBalancedTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory,
        ↪ UInt64RawLink* links, UInt64LinksHeader* header)
        {
            _links = links;
            _header = header;
            _memory = memory;
            _constants = memory.Constants;
        }

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

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

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

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

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

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

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

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

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

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

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

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

```
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
```

```
124         public ulong this[ulong index]
125         {
126             get
127             {
128                 var root = GetTreeRoot();
129                 if (index >= GetSize(root))
130                 {
131                     return 0UL;
132                 }
133                 while (root != 0UL)
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 + 1UL;
148                 }
149                 return 0UL; // 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 != 0UL)
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 0UL;
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 != 0UL)
190             {
191                 var @base = GetBasePartValue(root);
192                 if (@base <= link)
193                 {
194                     root = GetRightOrDefault(root);
195                 }
196                 else
197                 {
198                     totalRightIgnore += GetRightSize(root) + 1UL;
```

```
                    root = GetLeftOrDefault(root);
                }
            }
            root = GetTreeRoot();
            var totalLeftIgnore = 0UL;
            while (root != 0UL)
            {
                var @base = GetBasePartValue(root);
                if (@base >= link)
                {
                    root = GetLeftOrDefault(root);
                }
                else
                {
                    totalLeftIgnore += GetLeftSize(root) + 1UL;
                    root = GetRightOrDefault(root);
                }
            }
            return total - totalRightIgnore - totalLeftIgnore;
        }

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

        protected override void PrintNodeValue(ulong node, StringBuilder sb)
        {
            ref var link = ref _links[node];
            sb.Append(' ');
            sb.Append(link.Source);
            sb.Append('-');
            sb.Append('>');
            sb.Append(link.Target);
        }
    }
}
```

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

```
namespace Platform.Data.Doublets.ResizableDirectMemory
{
```

```
    internal struct UInt64LinksHeader
    {
        public ulong AllocatedLinks;
        public ulong ReservedLinks;
        public ulong FreeLinks;
        public ulong FirstFreeLink;
        public ulong FirstAsSource;
        public ulong FirstAsTarget;
        public ulong LastFreeLink;
        public ulong Reserved8;
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public unsafe class UInt64LinksSourcesAVLBalancedTreeMethods :
    ↪ UInt64LinksAVLBalancedTreeMethodsBase, ILinksTreeMethods<ulong>
    {
        internal UInt64LinksSourcesAVLBalancedTreeMethods(UInt64ResizableDirectMemoryLinks
        ↪ memory, UInt64RawLink* links, UInt64LinksHeader* header) : base(memory, links,
        ↪ header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
        ↪ _links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
        ↪ _links[node].RightAsSource;

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

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

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

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

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

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

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

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

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

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

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

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

```
        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

```
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;
    }
}
```

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

```
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;
```

```
72              }
73              SetPointers(_memory);
74              // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
75              _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(UInt64RawLink)) +
                ↪  sizeof(UInt64LinksHeader);
76              // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
77              _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(UInt64LinksHeader))
                ↪  / sizeof(UInt64RawLink));
78          }
79
80          [MethodImpl(MethodImplOptions.AggressiveInlining)]
81          public id Count(IList<id> restrictions)
82          {
83              // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
84              if (restrictions.Count == 0)
85              {
86                  return Total;
87              }
88              if (restrictions.Count == 1)
89              {
90                  var index = restrictions[Constants.IndexPart];
91                  if (index == Constants.Any)
92                  {
93                      return Total;
94                  }
95                  return Exists(index) ? 1UL : 0UL;
96              }
97              if (restrictions.Count == 2)
98              {
99                  var index = restrictions[Constants.IndexPart];
100                 var value = restrictions[1];
101                 if (index == Constants.Any)
102                 {
103                     if (value == Constants.Any)
104                     {
105                         return Total; // Any - как отсутствие ограничения
106                     }
107                     return _sourcesTreeMethods.CountUsages(value)
108                          + _targetsTreeMethods.CountUsages(value);
109                 }
110                 else
111                 {
112                     if (!Exists(index))
113                     {
114                         return 0;
115                     }
116                     if (value == Constants.Any)
117                     {
118                         return 1;
119                     }
120                     var storedLinkValue = GetLinkUnsafe(index);
121                     if (storedLinkValue->Source == value ||
122                         storedLinkValue->Target == value)
123                     {
124                         return 1;
125                     }
126                     return 0;
127                 }
128             }
129             if (restrictions.Count == 3)
130             {
131                 var index = restrictions[Constants.IndexPart];
132                 var source = restrictions[Constants.SourcePart];
133                 var target = restrictions[Constants.TargetPart];
134                 if (index == Constants.Any)
135                 {
136                     if (source == Constants.Any && target == Constants.Any)
137                     {
138                         return Total;
139                     }
140                     else if (source == Constants.Any)
141                     {
142                         return _targetsTreeMethods.CountUsages(target);
143                     }
144                     else if (target == Constants.Any)
145                     {
146                         return _sourcesTreeMethods.CountUsages(source);
147                     }
```

```
                    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)
```

```
226                 {
227                     var index = restrictions[Constants.IndexPart];
228                     var value = restrictions[1];
229                     if (index == Constants.Any)
230                     {
231                         if (value == Constants.Any)
232                         {
233                             return Each(handler, ArrayPool<ulong>.Empty);
234                         }
235                         if (Each(handler, new[] { index, value, Constants.Any }) == Constants.Break)
236                         {
237                             return Constants.Break;
238                         }
239                         return Each(handler, new[] { index, Constants.Any, value });
240                     }
241                     else
242                     {
243                         if (!Exists(index))
244                         {
245                             return Constants.Continue;
246                         }
247                         if (value == Constants.Any)
248                         {
249                             return handler(GetLinkStruct(index));
250                         }
251                         var storedLinkValue = GetLinkUnsafe(index);
252                         if (storedLinkValue->Source == value ||
253                             storedLinkValue->Target == value)
254                         {
255                             return handler(GetLinkStruct(index));
256                         }
257                         return Constants.Continue;
258                     }
259                 }
260                 if (restrictions.Count == 3)
261                 {
262                     var index = restrictions[Constants.IndexPart];
263                     var source = restrictions[Constants.SourcePart];
264                     var target = restrictions[Constants.TargetPart];
265                     if (index == Constants.Any)
266                     {
267                         if (source == Constants.Any && target == Constants.Any)
268                         {
269                             return Each(handler, ArrayPool<ulong>.Empty);
270                         }
271                         else if (source == Constants.Any)
272                         {
273                             return _targetsTreeMethods.EachUsage(target, handler);
274                         }
275                         else if (target == Constants.Any)
276                         {
277                             return _sourcesTreeMethods.EachUsage(source, handler);
278                         }
279                         else //if(source != Any && target != Any)
280                         {
281                             var link = _sourcesTreeMethods.Search(source, target);
282                             return link == Constants.Null ? Constants.Continue :
                                ↪ handler(GetLinkStruct(link));
283                         }
284                     }
285                     else
286                     {
287                         if (!Exists(index))
288                         {
289                             return Constants.Continue;
290                         }
291                         if (source == Constants.Any && target == Constants.Any)
292                         {
293                             return handler(GetLinkStruct(index));
294                         }
295                         var storedLinkValue = GetLinkUnsafe(index);
296                         if (source != Constants.Any && target != Constants.Any)
297                         {
298                             if (storedLinkValue->Source == source &&
299                                 storedLinkValue->Target == target)
300                             {
301                                 return handler(GetLinkStruct(index));
302                             }
```

```
                        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);
        }

```

```
        [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
            {
```

```
                _header = (UInt64LinksHeader*)(void*)memory.Pointer;
                _links = (UInt64RawLink*)(void*)memory.Pointer;
                _sourcesTreeMethods = new UInt64LinksSourcesAVLBalancedTreeMethods(this, _links,
                ↪  _header);
                _targetsTreeMethods = new UInt64LinksTargetsAVLBalancedTreeMethods(this, _links,
                ↪  _header);
                _unusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
            }
        }

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

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

        #region Disposable

        protected override bool AllowMultipleDisposeCalls => true;

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                SetPointers(null);
                _memory.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

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

```
using Platform.Collections.Methods.Lists;
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public unsafe class UInt64UnusedLinksListMethods : CircularDoublyLinkedListMethods<ulong>,
    ↪  ILinksListMethods<ulong>
    {
        private readonly UInt64RawLink* _links;
        private readonly UInt64LinksHeader* _header;

        internal UInt64UnusedLinksListMethods(UInt64RawLink* links, UInt64LinksHeader* header)
        {
            _links = links;
            _header = header;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetPrevious(ulong element) => _links[element].Source;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetFirst(ulong element) => _header->FirstFreeLink = element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLast(ulong element) => _header->LastFreeLink = element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPrevious(ulong element, ulong previous) =>
        ↪  _links[element].Source = previous;
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetNext(ulong element, ulong next) => _links[element].Target =
        ↪  next;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(ulong size) => _header->FreeLinks = size;
    }
}
```

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

```
using Platform.Collections.Methods.Lists;
using Platform.Numbers;
using System.Runtime.CompilerServices;
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 UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
    ↪  ILinksListMethods<TLink>
    {
        private readonly byte* _links;
        private readonly byte* _header;

        public UnusedLinksListMethods(byte* links, byte* header)
        {
            _links = links;
            _header = header;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetFirst() => GetHeader().FirstFreeLink;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLast() => GetHeader().LastFreeLink;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetPrevious(TLink element) => GetLink(element).Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNext(TLink element) => GetLink(element).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize() => GetHeader().FreeLinks;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetFirst(TLink element) => GetHeader().FirstFreeLink = element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLast(TLink element) => GetHeader().LastFreeLink = element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPrevious(TLink element, TLink previous) =>
        ↪  GetLink(element).Source = previous;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetNext(TLink element, TLink next) => GetLink(element).Target =
        ↪  next;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink size) => GetHeader().FreeLinks = size;
    }
}
```

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

```
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 ArrayExtensions
    {
        public static IList<TLink> ConvertToRestrictionsValues<TLink>(this TLink[] array)
        {
            var restrictions = new TLink[array.Length + 1];
            Array.Copy(array, 0, restrictions, 1, array.Length);
            return restrictions;
        }
    }
}
```

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

```
using System.Collections.Generic;

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

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

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

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

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

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

```
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)
```

```
        {
            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 }
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    public abstract class LinksListToSequenceConverterBase<TLink> : IConverter<IList<TLink>,
    ↪  TLink>
    {
        protected readonly ILinks<TLink> Links;
        public LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
        public abstract TLink Convert(IList<TLink> source);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;

        public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
        ↪  sequenceToItsLocalElementLevelsConverter) : base(links)
            => _sequenceToItsLocalElementLevelsConverter =
            ↪  sequenceToItsLocalElementLevelsConverter;

        public override TLink Convert(IList<TLink> sequence)
        {
            var length = sequence.Count;
            if (length == 1)
            {
                return sequence[0];
            }
            var links = Links;
            if (length == 2)
            {
                return links.GetOrCreate(sequence[0], sequence[1]);
            }
            sequence = sequence.ToArray();
            var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
            while (length > 2)
            {
                var levelRepeat = 1;
                var currentLevel = levels[0];
                var previousLevel = levels[0];
                var skipOnce = false;
                var w = 0;
                for (var i = 1; i < length; i++)
                {
                    if (_equalityComparer.Equals(currentLevel, levels[i]))
                    {
                        levelRepeat++;
                        skipOnce = false;
                        if (levelRepeat == 2)
                        {
                            sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
                            var newLevel = i >= length - 1 ?
                                GetPreviousLowerThanCurrentOrCurrent(previousLevel,
                                ↪  currentLevel) :
                                i < 2 ?
                                GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
                                GetGreatestNeigbourLowerThanCurrentOrCurrent(previousLevel,
                                ↪  currentLevel, levels[i + 1]);
                            levels[w] = newLevel;
                            previousLevel = currentLevel;
                            w++;
                            levelRepeat = 0;
```

```
                            skipOnce = true;
                        }
                        else if (i == length - 1)
                        {
                            sequence[w] = sequence[i];
                            levels[w] = levels[i];
                            w++;
                        }
                    }
                    else
                    {
                        currentLevel = levels[i];
                        levelRepeat = 1;
                        if (skipOnce)
                        {
                            skipOnce = false;
                        }
                        else
                        {
                            sequence[w] = sequence[i - 1];
                            levels[w] = levels[i - 1];
                            previousLevel = levels[w];
                            w++;
                        }
                        if (i == length - 1)
                        {
                            sequence[w] = sequence[i];
                            levels[w] = levels[i];
                            w++;
                        }
                    }
                }
                length = w;
            }
            return links.GetOrCreate(sequence[0], sequence[1]);
        }

        private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink current, TLink next)
        {
            return _comparer.Compare(previous, next) > 0
                ? _comparer.Compare(previous, current) < 0 ? previous : current
                : _comparer.Compare(next, current) < 0 ? next : current;
        }

        private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) => _comparer.Compare(next, current) < 0 ? next : current;

        private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current) => _comparer.Compare(previous, current) < 0 ? previous : current;
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>, IConverter<IList<TLink>>
    {
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;

        public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links, IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links) => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;

        public IList<TLink> Convert(IList<TLink> sequence)
        {
            var levels = new TLink[sequence.Count];
            levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
            for (var i = 1; i < sequence.Count - 1; i++)
            {
                var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
```

```
                var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
                levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
            }
            levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
            ↪  sequence[sequence.Count - 1]);
            return levels;
        }

        public TLink GetFrequencyNumber(TLink source, TLink target) =>
        ↪  _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
    }
}
```

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

```
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
{
    public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
    ↪  ICriterionMatcher<TLink>
    {
        public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
        public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
{
    public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

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

        public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
        {
            _links = links;
            _sequenceMarkerLink = sequenceMarkerLink;
        }

        public bool IsMatched(TLink sequenceCandidate)
            => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
            || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
            ↪  sequenceCandidate), _links.Constants.Null);
    }
}
```

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

```
using System.Collections.Generic;
using Platform.Collections.Stacks;
using Platform.Data.Doublets.Sequences.HeightProviders;
using Platform.Data.Sequences;

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

namespace Platform.Data.Doublets.Sequences
{
    public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceAppender<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

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

        public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
        ↪  ISequenceHeightProvider<TLink> heightProvider)
            : base(links)
```

```
        {
            _stack = stack;
            _heightProvider = heightProvider;
        }

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

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

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

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

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

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

```
using System;
using System.Linq;
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Collections;
using Platform.Collections.Lists;
using Platform.Collections.Segments;
using Platform.Collections.Segments.Walkers;
using Platform.Singletons;
using Platform.Numbers;
using Platform.Data.Doublets.Unicode;

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

namespace Platform.Data.Doublets.Sequences
{
    public class DuplicateSegmentsProvider<TLink> :
    ↪  DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
    ↪  IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
    {
        private readonly ILinks<TLink> _links;
        private readonly ILinks<TLink> _sequences;
        private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
        private BitString _visited;

```

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

```
90
91          protected override void OnDublicateFound(Segment<TLink> segment)
92          {
93              var duplicates = CollectDuplicatesForSegment(segment);
94              if (duplicates.Count > 1)
95              {
96                  _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                        ↪ duplicates));
97              }
98          }
99
100         private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
101         {
102             var duplicates = new List<TLink>();
103             var readAsElement = new HashSet<TLink>();
104             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

```
1  using System;
2  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.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();
```

```
            }
            else
            {
                var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
                data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
                if (!_equalityComparer.Equals(link, default))
                {
                    data.Frequency = Arithmetic.Add(data.Frequency,
                    ↪  _frequencyCounter.Count(link));
                }
                _doubletsCache.Add(doublet, data);
            }
            return data;
        }

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

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

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

```
using System.Runtime.CompilerServices;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    public class LinkFrequency<TLink>
    {
        public TLink Frequency { get; set; }
        public TLink Link { get; set; }

        public LinkFrequency(TLink frequency, TLink link)
        {
            Frequency = frequency;
            Link = link;
        }

        public LinkFrequency() { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

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

```
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

```
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

```
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;
```

```
        }

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

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

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

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

```
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 TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
    {
        private readonly ILinks<TLink> _links;
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

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

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

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

```
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.Counters
{
    public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
    ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>
    {
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

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

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

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

```
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 TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
    {
        private readonly ILinks<TLink> _links;
        public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
        public TLink Count(TLink symbol) => new
        ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
    }
}
```

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

```
using System.Collections.Generic;
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.Counters
{
    public class TotalSequenceSymbolFrequencyOneOffCounter<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 _symbol;
        protected readonly HashSet<TLink> _visits;
        protected TLink _total;

        public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
        {
            _links = links;
            _symbol = symbol;
            _visits = new HashSet<TLink>();
            _total = default;
        }

        public TLink Count()
        {
            if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
            {
                return _total;
            }
            CountCore(_symbol);
            return _total;
        }

        private void CountCore(TLink link)
        {
            var any = _links.Constants.Any;
            if (_equalityComparer.Equals(_links.Count(any, link), default))
            {
                CountSequenceSymbolFrequency(link);
            }
            else
            {
                _links.Each(EachElementHandler, any, link);
            }
        }

        protected virtual void CountSequenceSymbolFrequency(TLink link)
        {
            var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
            ↪  link, _symbol);
            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
        }

        private TLink EachElementHandler(IList<TLink> doublet)
        {
            var constants = _links.Constants;
            var doubletIndex = doublet[constants.IndexPart];
            if (_visits.Add(doubletIndex))
            {
```

```
                CountCore(doubletIndex);
            }
            return constants.Continue;
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceHeightProvider<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _heightPropertyMarker;
        private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
        private readonly IConverter<TLink> _addressToUnaryNumberConverter;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;
        private readonly IPropertiesOperator<TLink, TLink, TLink> _propertyOperator;

        public CachedSequenceHeightProvider(
            ILinks<TLink> links,
            ISequenceHeightProvider<TLink> baseHeightProvider,
            IConverter<TLink> addressToUnaryNumberConverter,
            IConverter<TLink> unaryNumberToAddressConverter,
            TLink heightPropertyMarker,
            IPropertiesOperator<TLink, TLink, TLink> propertyOperator)
            : base(links)
        {
            _heightPropertyMarker = heightPropertyMarker;
            _baseHeightProvider = baseHeightProvider;
            _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
            _propertyOperator = propertyOperator;
        }

        public TLink Get(TLink sequence)
        {
            TLink height;
            var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
            if (_equalityComparer.Equals(heightValue, default))
            {
                height = _baseHeightProvider.Get(sequence);
                heightValue = _addressToUnaryNumberConverter.Convert(height);
                _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
            }
            else
            {
                height = _unaryNumberToAddressConverter.Convert(heightValue);
            }
            return height;
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceHeightProvider<TLink>
    {
        private readonly ICriterionMatcher<TLink> _elementMatcher;

        public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
        ↪  elementMatcher) : base(links) => _elementMatcher = elementMatcher;

        public TLink Get(TLink sequence)
        {
```

```
            var height = default(TLink);
            var pairOrElement = sequence;
            while (!_elementMatcher.IsMatched(pairOrElement))
            {
                pairOrElement = Links.GetTarget(pairOrElement);
                height = Arithmetic.Increment(height);
            }
            return height;
        }
    }
}
```

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

```
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
    {
    }
}
```

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

```
using Platform.Collections;
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 IListExtensions
    {
        public static TLink[] ExtractValues<TLink>(this IList<TLink> restrictions)
        {
            if(restrictions.IsNullOrEmpty() || restrictions.Count == 1)
            {
                return new TLink[0];
            }
            var values = new TLink[restrictions.Count - 1];
            for (int i = 1, j = 0; i < restrictions.Count; i++, j++)
            {
                values[j] = restrictions[i];
            }
            return values;
        }

        public static IList<TLink> ConvertToRestrictionsValues<TLink>(this IList<TLink> list)
        {
            var restrictions = new TLink[list.Count + 1];
            for (int i = 0, j = 1; i < list.Count; i++, j++)
            {
                restrictions[j] = list[i];
            }
            return restrictions;
        }
    }
}
```

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

```
using System.Collections.Generic;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly LinkFrequenciesCache<TLink> _cache;

        public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
        ↪  _cache = cache;

        public bool Add(IList<TLink> sequence)
```

```
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i]))) { }
            for (; i >= 1; i--)
            {
                _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }

        private bool IsIndexedWithIncrement(TLink source, TLink target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
            if (indexed)
            {
                _cache.IncrementFrequency(source, target);
            }
            return indexed;
        }

        public bool MightContain(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
            return indexed;
        }

        private bool IsIndexed(TLink source, TLink target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            return !_equalityComparer.Equals(frequency.Frequency, default);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>, ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;

        private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
        private readonly IIncrementer<TLink> _frequencyIncrementer;

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

        public override bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i]))) { }
            for (; i >= 1; i--)
            {
```

```
                Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
            }
            return indexed;
        }

        private bool IsIndexedWithIncrement(TLink source, TLink target)
        {
            var link = Links.SearchOrDefault(source, target);
            var indexed = !_equalityComparer.Equals(link, default);
            if (indexed)
            {
                Increment(link);
            }
            return indexed;
        }

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

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

```
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public interface ISequenceIndex<TLink>
    {
        /// <summary>
        /// Индексирует последовательность глобально, и возвращает значение,
        /// определяющие была ли запрошенная последовательность проиндексирована ранее.
        /// </summary>
        /// <param name="sequence">Последовательность для индексации.</param>
        bool Add(IList<TLink> sequence);

        bool MightContain(IList<TLink> sequence);
    }
}
```

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

```
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;

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

        public virtual bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed =
            ↪ !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪ default))) { }
            for (; i >= 1; i--)
            {
                Links.GetOrCreate(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }

        public virtual bool MightContain(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed =
            ↪ !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪ default))) { }
```

```
            return indexed;
        }
    }
}
```

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

```
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly ISynchronizedLinks<TLink> _links;

        public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;

        public bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            var links = _links.Unsync;
            _links.SyncRoot.ExecuteReadOperation(() =>
            {
                while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪  sequence[i]), default))) { }
            });
            if (!indexed)
            {
                _links.SyncRoot.ExecuteWriteOperation(() =>
                {
                    for (; i >= 1; i--)
                    {
                        links.GetOrCreate(sequence[i - 1], sequence[i]);
                    }
                });
            }
            return indexed;
        }

        public bool MightContain(IList<TLink> sequence)
        {
            var links = _links.Unsync;
            return _links.SyncRoot.ExecuteReadOperation(() =>
            {
                var indexed = true;
                var i = sequence.Count;
                while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪  sequence[i]), default))) { }
                return indexed;
            });
        }
    }
}
```

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

```
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class Unindex<TLink> : ISequenceIndex<TLink>
    {
        public virtual bool Add(IList<TLink> sequence) => false;

        public virtual bool MightContain(IList<TLink> sequence) => true;
    }
}
```

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

```
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.Sequences
{
    public class ListFiller<TElement, TReturnConstant>
    {
        protected readonly List<TElement> _list;
        protected readonly TReturnConstant _returnConstant;

        public ListFiller(List<TElement> list, TReturnConstant returnConstant)
        {
            _list = list;
            _returnConstant = returnConstant;
        }

        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

```

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.
```

```
20      /// </summary>
21      /// <remarks>
22      /// Обязательно реализовать атомарность каждого публичного метода.
23      ///
24      /// TODO:
25      ///
26      /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
27      /// через естественную группировку по unicode типам, все whitespace вместе, все символы
        ↪  вместе, все числа вместе и т.п.
28      /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
        ↪  графа)
29      ///
30      /// 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)
106                 {
107                     return linkContents.Target;
108                 }
109                 if (linkContents.Target == Options.SequenceMarkerLink)
110                 {
111                     return linkContents.Source;
112                 }
113             }
114             return sequence;
115         }
116
117         #region Count
118
119         public LinkIndex Count(IList<LinkIndex> restrictions)
120         {
121             if (restrictions.IsNullOrEmpty())
122             {
123                 return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
124             }
125             if (restrictions.Count == 1) // Первая связь это адрес
126             {
127                 var sequenceIndex = restrictions[0];
128                 if (sequenceIndex == Constants.Null)
129                 {
130                     return 0;
131                 }
132                 if (sequenceIndex == Constants.Any)
133                 {
134                     return Count(null);
135                 }
136                 if (Options.UseSequenceMarker)
137                 {
138                     return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
139                 }
140                 return Links.Exists(sequenceIndex) ? 1UL : 0;
141             }
142             throw new NotImplementedException();
143         }
144
145         private LinkIndex CountUsages(params LinkIndex[] restrictions)
146         {
147             if (restrictions.Length == 0)
148             {
149                 return 0;
150             }
151             if (restrictions.Length == 1) // Первая связь это адрес
152             {
153                 if (restrictions[0] == Constants.Null)
154                 {
155                     return 0;
156                 }
157                 if (Options.UseSequenceMarker)
158                 {
159                     var elementsLink = GetSequenceElements(restrictions[0]);
160                     var sequenceLink = GetSequenceByElements(elementsLink);
161                     if (sequenceLink != Constants.Null)
162                     {
163                         return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
164                     }
165                     return Links.Count(elementsLink);
166                 }
167                 return Links.Count(restrictions[0]);
168             }
169             throw new NotImplementedException();
170         }
```

```
171
172         #endregion
173
174         #region Create
175
176         public LinkIndex Create(IList<LinkIndex> restrictions)
177         {
178             return _sync.ExecuteWriteOperation(() =>
179             {
180                 if (restrictions.IsNullOrEmpty())
181                 {
182                     return Constants.Null;
183                 }
184                 Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
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);
259             }
260             else if (restrictions.Count == 2)
261             {
262                 throw new NotImplementedException();
263             }
264             else if (restrictions.Count == 3)
265             {
266                 return Links.Unsync.Each(handler, restrictions);
267             }
268             else
269             {
270                 var sequence = restrictions.ExtractValues();
271                 if (Options.UseIndex && !Options.Index.MightContain(sequence))
272                 {
273                     return Constants.Break;
274                 }
275                 return EachCore(handler, sequence);
276             }
277         });
278     }
279
280     private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
        ↪  values)
281     {
282         var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
283         // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
            ↪  Id.
284         Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
            ↪  (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
            ↪  matcher.HandleFullMatched;
285         //if (sequence.Length >= 2)
286         if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
287         {
288             return Constants.Break;
289         }
290         var last = values.Count - 2;
291         for (var i = 1; i < last; i++)
292         {
293             if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
                ↪  Constants.Continue)
294             {
295                 return Constants.Break;
296             }
297         }
298         if (values.Count >= 3)
299         {
300             if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
                ↪  != Constants.Continue)
301             {
302                 return Constants.Break;
303             }
304         }
305         return Constants.Continue;
306     }
307
308     private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
        ↪  left, LinkIndex right)
309     {
310         return Links.Unsync.Each(doublet =>
311         {
312             var doubletIndex = doublet[Constants.IndexPart];
313             if (StepRight(handler, doubletIndex, right) != Constants.Continue)
314             {
315                 return Constants.Break;
316             }
```

```
317                 if (left != doubletIndex)
318                 {
319                     return PartialStepRight(handler, doubletIndex, right);
320                 }
321                 return Constants.Continue;
322             }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
323         }
324 
325         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));
326 
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>
554         /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
        ↪  определить извне или в унаследованном классе
555         /// </remarks>
556         [MethodImpl(MethodImplOptions.AggressiveInlining)]
557         private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
        ↪  !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;
558 
559         private void ClearGarbage(LinkIndex link)
560         {
561             if (IsGarbage(link))
562             {
563                 var contents = new UInt64Link(Links.GetLink(link));
564                 Links.Unsync.Delete(link);
565                 ClearGarbage(contents.Source);
566                 ClearGarbage(contents.Target);
567             }
568         }
569 
570         #endregion
571 
572         #region Walkers
573 
574         public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
575         {
576             return _sync.ExecuteReadOperation(() =>
577             {
578                 var links = Links.Unsync;
579                 foreach (var part in Options.Walker.Walk(sequence))
580                 {
581                     if (!handler(part))
582                     {
583                         return false;
584                     }
585                 }
586                 return true;
587             });
588         }
589 
590         public class Matcher : RightSequenceWalker<LinkIndex>
591         {
592             private readonly Sequences _sequences;
593             private readonly IList<LinkIndex> _patternSequence;
594             private readonly HashSet<LinkIndex> _linksInSequence;
595             private readonly HashSet<LinkIndex> _results;
596             private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
597             private readonly HashSet<LinkIndex> _readAsElements;
598             private int _filterPosition;
599 
600             public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
            ↪  HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
            ↪  HashSet<LinkIndex> readAsElements = null)
601                 : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
602             {
603                 _sequences = sequences;
604                 _patternSequence = patternSequence;
605                 _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                ↪  Links.Constants.Any && x != ZeroOrMany));
606                 _results = results;
607                 _stopableHandler = stopableHandler;
608                 _readAsElements = readAsElements;
609             }
610 
611             protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
            ↪  (_readAsElements != null && _readAsElements.Contains(link)) ||
            ↪  _linksInSequence.Contains(link);
612 
613             public bool FullMatch(LinkIndex sequenceToMatch)
```

```
614             {
615                 _filterPosition = 0;
616                 foreach (var part in Walk(sequenceToMatch))
617                 {
618                     if (!FullMatchCore(part))
619                     {
620                         break;
621                     }
622                 }
623                 return _filterPosition == _patternSequence.Count;
624             }
625 
626             private bool FullMatchCore(LinkIndex element)
627             {
628                 if (_filterPosition == _patternSequence.Count)
629                 {
630                     _filterPosition = -2; // Длиннее чем нужно
631                     return false;
632                 }
633                 if (_patternSequence[_filterPosition] != Links.Constants.Any
634                  && element != _patternSequence[_filterPosition])
635                 {
636                     _filterPosition = -1;
637                     return false; // Начинается/Продолжается иначе
638                 }
639                 _filterPosition++;
640                 return true;
641             }
642 
643             public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
644             {
645                 var sequenceToMatch = restrictions[Links.Constants.IndexPart];
646                 if (FullMatch(sequenceToMatch))
647                 {
648                     _results.Add(sequenceToMatch);
649                 }
650             }
651 
652             public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
653             {
654                 var sequenceToMatch = restrictions[Links.Constants.IndexPart];
655                 if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
656                 {
657                     return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
658                 }
659                 return Links.Constants.Continue;
660             }
661 
662             public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
663             {
664                 var sequenceToMatch = restrictions[Links.Constants.IndexPart];
665                 var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
666                 if (sequence != Links.Constants.Null && FullMatch(sequenceToMatch) &&
                    ↪ _results.Add(sequenceToMatch))
667                 {
668                     return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
669                 }
670                 return Links.Constants.Continue;
671             }
672 
673             /// <remarks>
674             /// TODO: Add support for LinksConstants.Any
675             /// </remarks>
676             public bool PartialMatch(LinkIndex sequenceToMatch)
677             {
678                 _filterPosition = -1;
679                 foreach (var part in Walk(sequenceToMatch))
680                 {
681                     if (!PartialMatchCore(part))
682                     {
683                         break;
684                     }
685                 }
686                 return _filterPosition == _patternSequence.Count - 1;
687             }
688 
689             private bool PartialMatchCore(LinkIndex element)
690             {
691                 if (_filterPosition == (_patternSequence.Count - 1))
```

```
                {
                    return false; // Нашлось
                }
                if (_filterPosition >= 0)
                {
                    if (element == _patternSequence[_filterPosition + 1])
                    {
                        _filterPosition++;
                    }
                    else
                    {
                        _filterPosition = -1;
                    }
                }
                if (_filterPosition < 0)
                {
                    if (element == _patternSequence[0])
                    {
                        _filterPosition = 0;
                    }
                }
                return true; // Ищем дальше
            }

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

            public LinkIndex HandlePartialMatched(IList<LinkIndex> restrictions)
            {
                var sequenceToMatch = restrictions[Links.Constants.IndexPart];
                if (PartialMatch(sequenceToMatch))
                {
                    return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
                }
                return Links.Constants.Continue;
            }

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

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

        #endregion
    }
}
```

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

```
using System;
using LinkIndex = System.UInt64;
using System.Collections.Generic;
using Stack = System.Collections.Generic.Stack<ulong>;
using System.Linq;
using System.Text;
using Platform.Collections;
```

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

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

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

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

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

        public List<ulong> CreateAllVariants1(params ulong[] sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
```

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

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

```
313         private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
314         {
315             Links.Unsync.Each(right, Constants.Any, doublet =>
316             {
317                 StepLeft(handler, left, doublet);
318                 if (right != doublet)
319                 {
320                     PartialStepLeft(handler, left, doublet);
321                 }
322                 return true;
323             });
324         }
325 
326         private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
327         {
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)
```

```
                    {
                        results.Add(doublet);
                    }
                    return results;
                }
                var linksInSequence = new HashSet<ulong>(sequence);
                void handler(IList<LinkIndex> result)
                {
                    var resultIndex = result[Links.Constants.IndexPart];
                    var filterPosition = 0;
                    StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
                    ↪  Links.Unsync.GetTarget,
                        x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                        ↪  x =>
                        {
                            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]);
                    }
                }
                return results;
            });
        }

        public const int MaxSequenceFormatSize = 200;

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

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

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

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

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

        private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
        ↪  Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
        ↪  LinkIndex[] knownElements)
```

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

```
                                    filterPosition = 0;
                                }
                            }
                            return true;
                        });
                    if (filterPosition == (sequence.Length - 1))
                    {
                        filteredResults.Add(result);
                    }
                }
                return filteredResults;
            }
            return new List<ulong>();
        });
    }

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

    public bool GetAllPartiallyMatchingSequences2(Func<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);

```

```
683         //            firstResults.IntersectWith(lastResults);
684
685         //            //for (var i = 0; i < sequence.Length; i++)
686         //            //    AllUsagesCore(sequence[i], results);
687
688         //            var filteredResults = new HashSet<ulong>();
689         //            var matcher = new Matcher(this, sequence, filteredResults, null);
690         //            matcher.AddAllPartialMatchedToResults(firstResults);
691         //            return filteredResults;
692         //        }
693
694         //        return new HashSet<ulong>();
695         //    });
696         //}
697
698         public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
699         {
700             return _sync.ExecuteReadOperation(() =>
701             {
702                 if (sequence.Length > 0)
703                 {
704                     Links.EnsureEachLinkIsAnyOrExists(sequence);
705                     var firstResults = new HashSet<ulong>();
706                     var lastResults = new HashSet<ulong>();
707                     var first = sequence.First(x => x != Constants.Any);
708                     var last = sequence.Last(x => x != Constants.Any);
709                     AllUsagesCore(first, firstResults);
710                     AllUsagesCore(last, lastResults);
711                     firstResults.IntersectWith(lastResults);
712                     //for (var i = 0; i < sequence.Length; i++)
713                     //    AllUsagesCore(sequence[i], results);
714                     var filteredResults = new HashSet<ulong>();
715                     var matcher = new Matcher(this, sequence, filteredResults, null);
716                     matcher.AddAllPartialMatchedToResults(firstResults);
717                     return filteredResults;
718                 }
719                 return new HashSet<ulong>();
720             });
721         }
722
723         public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
            ↪ IList<ulong> sequence)
724         {
725             return _sync.ExecuteReadOperation(() =>
726             {
727                 if (sequence.Count > 0)
728                 {
729                     Links.EnsureEachLinkExists(sequence);
730                     var results = new HashSet<LinkIndex>();
731                     //var nextResults = new HashSet<ulong>();
732                     //for (var i = 0; i < sequence.Length; i++)
733                     //{
734                     //    AllUsagesCore(sequence[i], nextResults);
735                     //    if (results.IsNullOrEmpty())
736                     //    {
737                     //        results = nextResults;
738                     //        nextResults = new HashSet<ulong>();
739                     //    }
740                     //    else
741                     //    {
742                     //        results.IntersectWith(nextResults);
743                     //        nextResults.Clear();
744                     //    }
745                     //}
746                     var collector1 = new AllUsagesCollector1(Links.Unsync, results);
747                     collector1.Collect(Links.Unsync.GetLink(sequence[0]));
748                     var next = new HashSet<ulong>();
749                     for (var i = 1; i < sequence.Count; i++)
750                     {
751                         var collector = new AllUsagesCollector1(Links.Unsync, next);
752                         collector.Collect(Links.Unsync.GetLink(sequence[i]));
753
754                         results.IntersectWith(next);
755                         next.Clear();
756                     }
757                     var filteredResults = new HashSet<ulong>();
758                     var matcher = new Matcher(this, sequence, filteredResults, null,
                        ↪ readAsElements);
```

```
759                     matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
                        ↪  x)); // OrderBy is a Hack
760                     return filteredResults;
761                 }
762                 return new HashSet<ulong>();
763             });
764         }
765 
766         // Does not work
767         //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
                ↪  params ulong[] sequence)
768         //{
769         //    var visited = new HashSet<ulong>();
770         //    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]);
```

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

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

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

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

```
976         public void CalculateAllUsages2(ulong[] totals)
977         {
978             var calculator = new AllUsagesCalculator2(Links, totals);
979             calculator.Calculate();
980         }
981 
982         private class AllUsagesCalculator
983         {
984             private readonly SynchronizedLinks<ulong> _links;
985             private readonly ulong[] _totals;
986 
987             public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
988             {
989                 _links = links;
990                 _totals = totals;
991             }
992 
993             public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
                ↪ CalculateCore);
994 
995             private bool CalculateCore(ulong link)
996             {
997                 if (_totals[link] == 0)
998                 {
999                     var total = 1UL;
1000                    _totals[link] = total;
1001                    var visitedChildren = new HashSet<ulong>();
1002                    bool linkCalculator(ulong child)
1003                    {
1004                        if (link != child && visitedChildren.Add(child))
1005                        {
1006                            total += _totals[child] == 0 ? 1 : _totals[child];
1007                        }
1008                        return true;
1009                    }
1010                    _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1011                    _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1012                    _totals[link] = total;
1013                }
1014                return true;
1015            }
1016        }
1017 
1018        private class AllUsagesCalculator2
1019        {
1020            private readonly SynchronizedLinks<ulong> _links;
1021            private readonly ulong[] _totals;
1022 
1023            public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1024            {
1025                _links = links;
1026                _totals = totals;
1027            }
1028 
1029            public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
                ↪ CalculateCore);
1030 
1031            private bool IsElement(ulong link)
1032            {
1033                //_linksInSequence.Contains(link) ||
1034                return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
                    ↪ link;
1035            }
1036 
1037            private bool CalculateCore(ulong link)
1038            {
1039                // TODO: Проработать защиту от зацикливания
1040                // Основано на SequenceWalker.WalkLeft
1041                Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1042                Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1043                Func<ulong, bool> isElement = IsElement;
1044                void visitLeaf(ulong parent)
1045                {
1046                    if (link != parent)
1047                    {
1048                        _totals[parent]++;
1049                    }
1050                }
1051                void visitNode(ulong parent)
```

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

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

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

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

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

            public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
            {
```

```
                _links = links;
                _usages = usages;
                _continue = _links.Constants.Continue;
            }

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

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

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

            public bool Collect(ulong link)
            {
                if (_usages.Add((long)link))
                {
                    _links.Each(link, _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);
        }

        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(() =>
```

```
1275                {
1276                    patternSequence = Simplify(patternSequence);
1277                    if (patternSequence.Length > 0)
1278                    {
1279                        EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1280                        var uniqueSequenceElements = new HashSet<ulong>();
1281                        for (var i = 0; i < patternSequence.Length; i++)
1282                        {
1283                            if (patternSequence[i] != Constants.Any && patternSequence[i] !=
                                ↪  ZeroOrMany)
1284                            {
1285                                uniqueSequenceElements.Add(patternSequence[i]);
1286                            }
1287                        }
1288                        var results = new HashSet<ulong>();
1289                        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
                ↪  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(() =>
```

```
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureEachLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector(Links, results);
                    collector1.Collect(linksToConnect[0]);
                    //AllUsagesCore(linksToConnect[0], results);
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var next = new HashSet<ulong>();
                        var collector = new AllUsagesIntersectingCollector(Links, results, next);
                        collector.Collect(linksToConnect[i]);
                        //AllUsagesCore(linksToConnect[i], next);
                        //results.IntersectWith(next);
                        results = next;
                    }
                }
                return results;
            });
        }

        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;
```

```
1427                //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1428                //if (zeroOrManyStepped && newZeroOrManyStepped)
1429                //    continue;
1430                //zeroOrManyStepped = newZeroOrManyStepped;
1431                if (sequence[i] == ZeroOrMany)
1432                {
1433                    if (zeroOrManyStepped)
1434                    {
1435                        continue;
1436                    }
1437                    zeroOrManyStepped = true;
1438                }
1439                else
1440                {
1441                    //if (zeroOrManyStepped) Is it efficient?
1442                    zeroOrManyStepped = false;
1443                }
1444                newSequence[j++] = sequence[i];
1445            }
1446            return newSequence;
1447        }
1448
1449        public static void TestSimplify()
1450        {
1451            var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
                ↪  ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1452            var simplifiedSequence = Simplify(sequence);
1453        }
1454
1455        public List<ulong> GetSimilarSequences() => new List<ulong>();
1456
1457        public void Prediction()
1458        {
1459            //_links
1460            //sequences
1461        }
1462
1463        #region From Triplets
1464
1465        //public static void DeleteSequence(Link sequence)
1466        //{
1467        //}
1468
1469        public List<ulong> CollectMatchingSequences(ulong[] links)
1470        {
1471            if (links.Length == 1)
1472            {
1473                throw new Exception("Подпоследовательности с одним элементом не
                    ↪  поддерживаются.");
1474            }
1475            var leftBound = 0;
1476            var rightBound = links.Length - 1;
1477            var left = links[leftBound++];
1478            var right = links[rightBound--];
1479            var results = new List<ulong>();
1480            CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1481            return results;
1482        }
1483
1484        private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
        ↪  middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1485        {
1486            var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1487            var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1488            if (leftLinkTotalReferers <= rightLinkTotalReferers)
1489            {
1490                var nextLeftLink = middleLinks[leftBound];
1491                var elements = GetRightElements(leftLink, nextLeftLink);
1492                if (leftBound <= rightBound)
1493                {
1494                    for (var i = elements.Length - 1; i >= 0; i--)
1495                    {
1496                        var element = elements[i];
1497                        if (element != 0)
1498                        {
1499                            CollectMatchingSequences(element, leftBound + 1, middleLinks,
                                ↪  rightLink, rightBound, ref results);
1500                        }
1501                    }
```

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

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

        public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
        {
            var added = 0;
            Links.Each(startLink, Constants.Any, couple =>
            {
                if (couple != startLink)
                {
                    var coupleTarget = Links.GetTarget(couple);
                    if (coupleTarget == rightLink)
                    {
                        result[offset] = couple;
                        if (++added == 2)
                        {
```

```
                        return false;
                    }
                }
                else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
                ↪  == Net.And &&
                {
                    result[offset + 1] = couple;
                    if (++added == 2)
                    {
                        return false;
                    }
                }
            }
            return true;
        });
        return added > 0;
    }

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

```
1656            private readonly HashSet<LinkIndex> _linksInSequence;
1657            private readonly HashSet<LinkIndex> _results;
1658
1659            #region Pattern Match
1660
1661            enum PatternBlockType
1662            {
1663                Undefined,
1664                Gap,
1665                Elements
1666            }
1667
1668            struct PatternBlock
1669            {
1670                public PatternBlockType Type;
1671                public long Start;
1672                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                        }
```

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

//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
                    {
                        _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

```
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());
        }

        public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
        {
            var list = new List<TLink>();
            var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
            sequences.Each(filler.AddAllValuesAndReturnConstant, new
            ↪ LinkAddress<TLink>(sequence));
            return list;
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Collections.Stacks;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.CreteriaMatchers;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences.Indexes;

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

namespace Platform.Data.Doublets.Sequences
{
    public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
    ↪ ILinks<TLink> must contain GetConstants function.
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;

        public TLink SequenceMarkerLink { get; set; }
        public bool UseCascadeUpdate { get; set; }
        public bool UseCascadeDelete { get; set; }
        public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
        public bool UseSequenceMarker { get; set; }
        public bool UseCompression { get; set; }
        public bool UseGarbageCollection { get; set; }
        public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
        public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }

        public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher { get; set; }
        public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
        public ISequenceIndex<TLink> Index { get; set; }
        public ISequenceWalker<TLink> Walker { get; set; }
        public bool ReadFullSequence { get; set; }

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

        public void InitOptions(ISynchronizedLinks<TLink> links)
        {
            if (UseSequenceMarker)
            {
                if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
                {
                    SequenceMarkerLink = links.CreatePoint();
                }
```

```
                else
                {
                    if (!links.Exists(SequenceMarkerLink))
                    {
                        var link = links.CreatePoint();
                        if (!_equalityComparer.Equals(link, SequenceMarkerLink))
                        {
                            throw new InvalidOperationException("Cannot recreate sequence marker link.");
                        }
                    }
                }
                if (MarkedSequenceMatcher == null)
                {
                    MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links, SequenceMarkerLink);
                }
            }
            var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
            if (UseCompression)
            {
                if (LinksToSequenceConverter == null)
                {
                    ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
                    if (UseSequenceMarker)
                    {
                        totalSequenceSymbolFrequencyCounter = new TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links, MarkedSequenceMatcher);
                    }
                    else
                    {
                        totalSequenceSymbolFrequencyCounter = new TotalSequenceSymbolFrequencyCounter<TLink>(links);
                    }
                    var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links, totalSequenceSymbolFrequencyCounter);
                    var compressingConverter = new CompressingConverter<TLink>(links, balancedVariantConverter, doubletFrequenciesCache);
                    LinksToSequenceConverter = compressingConverter;
                }
            }
            else
            {
                if (LinksToSequenceConverter == null)
                {
                    LinksToSequenceConverter = balancedVariantConverter;
                }
            }
            if (UseIndex && Index == null)
            {
                Index = new SequenceIndex<TLink>(links);
            }
            if (Walker == null)
            {
                Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
            }
        }

        public void ValidateOptions()
        {
            if (UseGarbageCollection && !UseSequenceMarker)
            {
                throw new NotSupportedException("To use garbage collection UseSequenceMarker option must be on.");
            }
        }
    }
}
```

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

```
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.Sequences
{
```

```
    public class SetFiller<TElement, TReturnConstant>
    {
        protected readonly ISet<TElement> _set;
        protected readonly TReturnConstant _returnConstant;

        public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
        {
            _set = set;
            _returnConstant = returnConstant;
        }

        public SetFiller(ISet<TElement> set) : this(set, default) { }

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

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

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

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

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

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

```
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public interface ISequenceWalker<TLink>
    {
        IEnumerable<TLink> Walk(TLink sequence);
    }
}
```

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

```
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 LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links, stack, isElement) { }

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

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

```

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

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

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

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

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

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

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

        private readonly Func<TLink, bool> _isElement;

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

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

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

        public TLink[] ToArray(TLink sequence)
        {
            var length = 1;
            var array = new TLink[length];
            array[0] = sequence;
            if (_isElement(sequence))
            {
                return array;
            }
            bool hasElements;
            do
            {
                length *= 2;
#if USEARRAYPOOL
                var nextArray = ArrayPool.Allocate<ulong>(length);
#else
                var nextArray = new TLink[length];
#endif
                hasElements = false;
                for (var i = 0; i < array.Length; i++)
                {
                    var candidate = array[i];
                    if (_equalityComparer.Equals(array[i], default))
                    {
                        continue;
                    }
                    var doubletOffset = i * 2;
                    if (_isElement(candidate))
                    {
```

```
                        nextArray[doubletOffset] = candidate;
                    }
                    else
                    {
                        var link = Links.GetLink(candidate);
                        var linkSource = Links.GetSource(link);
                        var linkTarget = Links.GetTarget(link);
                        nextArray[doubletOffset] = linkSource;
                        nextArray[doubletOffset + 1] = linkTarget;
                        if (!hasElements)
                        {
                            hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
                        }
                    }
                }
#if USEARRAYPOOL
                if (array.Length > 1)
                {
                    ArrayPool.Free(array);
                }
#endif
                array = nextArray;
            }
            while (hasElements);
            var filledElementsCount = CountFilledElements(array);
            if (filledElementsCount == array.Length)
            {
                return array;
            }
            else
            {
                return CopyFilledElements(array, filledElementsCount);
            }
        }

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

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

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

```
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);

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

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

```
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 abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceWalker<TLink>
    {
        private readonly IStack<TLink> _stack;
        private readonly Func<TLink, bool> _isElement;

        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links)
        {
            _stack = stack;
            _isElement = isElement;
        }

        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
        ↪  stack, links.IsPartialPoint)
        {
        }

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

```
                        {
                            yield return output;
                        }
                        element = GetNextElementAfterPop(element);
                    }
                    else
                    {
                        _stack.Push(element);
                        element = GetNextElementAfterPush(element);
                    }
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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

```
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

```
#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

```
using System;
using System.Collections.Generic;
using Platform.Data.Doublets;
using Platform.Threading.Synchronization;

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

namespace Platform.Data.Doublets
{
    /// <remarks>
    /// TODO: Autogeneration of synchronized wrapper (decorator).
    /// TODO: Try to unfold code of each method using IL generation for performance improvements.
    /// TODO: Or even to unfold multiple layers of implementations.
    /// </remarks>
    public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
    {
        public LinksConstants<TLinkAddress> Constants { get; }
        public ISynchronization SyncRoot { get; }
        public ILinks<TLinkAddress> Sync { get; }
        public ILinks<TLinkAddress> Unsync { get; }

        public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
        ↪  ReaderWriterLockSynchronization(), links) { }

        public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
        {
            SyncRoot = synchronization;
            Sync = this;
            Unsync = links;
            Constants = links.Constants;
        }

        public TLinkAddress Count(IList<TLinkAddress> restriction) =>
        ↪  SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
        public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
        ↪  IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
        ↪  restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
        public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
        ↪  SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
        public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
        ↪  substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
        ↪  Unsync.Update);
        public void Delete(IList<TLinkAddress> restrictions) =>
        ↪  SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);

        //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
        ↪  IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
        //{
        //    if (restriction != null && substitution != null &&
        ↪  !substitution.EqualTo(restriction))
        //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
        ↪  substitution, substitutedHandler, Unsync.Trigger);

        //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
        ↪  substitutedHandler, Unsync.Trigger);
        //}
    }
}
```

./Platform.Data.Doublets/UInt64Link.cs

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

```
using Platform.Singletons;
using Platform.Collections.Lists;

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

namespace Platform.Data.Doublets
{
    /// <summary>
    /// Структура описывающая уникальную связь.
    /// </summary>
    public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
    {
        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();

```

```
76          public static implicit operator UInt64Link(ulong[] linkArray) => new
            ↪  UInt64Link(linkArray);
77
78          public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
            ↪  : ToString(Index, Source, Target);
79
80          #region IList
81
82          public ulong this[int index]
83          {
84              get
85              {
86                  Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
                    ↪  nameof(index));
87                  if (index == _constants.IndexPart)
88                  {
89                      return Index;
90                  }
91                  if (index == _constants.SourcePart)
92                  {
93                      return Source;
94                  }
95                  if (index == _constants.TargetPart)
96                  {
97                      return Target;
98                  }
99                  throw new NotSupportedException(); // Impossible path due to
                    ↪  Ensure.ArgumentInRange
100             }
101             set => throw new NotSupportedException();
102         }
103
104         public int Count => Length;
105
106         public bool IsReadOnly => true;
107
108         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
109
110         public IEnumerator<ulong> GetEnumerator()
111         {
112             yield return Index;
113             yield return Source;
114             yield return Target;
115         }
116
117         public void Add(ulong item) => throw new NotSupportedException();
118
119         public void Clear() => throw new NotSupportedException();
120
121         public bool Contains(ulong item) => IndexOf(item) >= 0;
122
123         public void CopyTo(ulong[] array, int arrayIndex)
124         {
125             Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
126             Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                ↪  nameof(arrayIndex));
127             if (arrayIndex + Length > array.Length)
128             {
129                 throw new ArgumentException();
130             }
131             array[arrayIndex++] = Index;
132             array[arrayIndex++] = Source;
133             array[arrayIndex] = Target;
134         }
135
136         public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
137
138         public int IndexOf(ulong item)
139         {
140             if (Index == item)
141             {
142                 return _constants.IndexPart;
143             }
144             if (Source == item)
145             {
146                 return _constants.SourcePart;
147             }
148             if (Target == item)
149             {
```

```
                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

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

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

./Platform.Data.Doublets/UInt64LinksExtensions.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using Platform.Singletons;
using Platform.Data.Exceptions;
using Platform.Data.Doublets.Unicode;

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

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

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

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

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

```

```
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              }
62              return false;
63          }
64
65          public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
            ↪  Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
66          {
67              var sb = new StringBuilder();
68              var visited = new HashSet<ulong>();
69              links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
                ↪  innerSb.Append(link.Index), renderIndex, renderDebug);
70              return sb.ToString();
71          }
72
73          public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
            ↪  Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
            ↪  bool renderIndex = false, bool renderDebug = false)
74          {
75              var sb = new StringBuilder();
76              var visited = new HashSet<ulong>();
77              links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
                ↪  renderDebug);
78              return sb.ToString();
79          }
80
81          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)
82          {
83              if (sb == null)
84              {
85                  throw new ArgumentNullException(nameof(sb));
86              }
87              if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
                ↪  Constants.Itself)
88              {
89                  return;
90              }
91              if (links.Exists(linkIndex))
92              {
93                  if (visited.Add(linkIndex))
94                  {
95                      sb.Append('(');
96                      var link = new UInt64Link(links.GetLink(linkIndex));
97                      if (renderIndex)
98                      {
99                          sb.Append(link.Index);
100                         sb.Append(':');
101                     }
102                     if (link.Source == link.Index)
103                     {
104                         sb.Append(link.Index);
105                     }
106                     else
107                     {
108                         var source = new UInt64Link(links.GetLink(link.Source));
109                         if (isElement(source))
110                         {
111                             appendElement(sb, source);
112                         }
113                         else
114                         {
115                             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);
                        }
                        else
                        {
                            links.AppendStructure(sb, visited, target.Index, isElement,
                            ↪  appendElement, renderIndex);
                        }
                    }
                    sb.Append(')');
                }
                else
                {
                    if (renderDebug)
                    {
                        sb.Append('*');
                    }
                    sb.Append(linkIndex);
                }
            }
            else
            {
                if (renderDebug)
                {
                    sb.Append('~');
                }
                sb.Append(linkIndex);
            }
        }
    }
}
```

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.IO;
using System.Runtime.CompilerServices;
using System.Threading;
using System.Threading.Tasks;
using Platform.Disposables;
using Platform.Timestamps;
using Platform.Unsafe;
using Platform.IO;
using Platform.Data.Doublets.Decorators;
using Platform.Exceptions;

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

namespace Platform.Data.Doublets
{
    public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
    {
        /// <remarks>
        /// Альтернативные варианты хранения трансформации (элемента транзакции):
        ///
        /// private enum TransitionType
        /// {
        ///     Creation,
        ///     UpdateOf,
        ///     UpdateTo,
        ///     Deletion
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public ulong TransactionId;
        ///     public UniqueTimestamp Timestamp;
```

```
        ///     public TransactionItemType Type;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
        /// }
        ///
        /// Или
        ///
        /// public struct TransitionHeader
        /// {
        ///     public ulong TransactionIdCombined;
        ///     public ulong TimestampCombined;
        ///
        ///     public ulong TransactionId
        ///     {
        ///         get
        ///         {
        ///             return (ulong) mask &amp;amp; TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public UniqueTimestamp Timestamp
        ///     {
        ///         get
        ///         {
        ///             return (UniqueTimestamp)mask &amp;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)
            {
            }

```

```
112             public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
                ↪  {After}";
113         }
114
115         /// <remarks>
116         /// Другие варианты реализации транзакций (атомарности):
117         ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
                ↪  Target)) и индексов.
118         ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
                ↪  потребуется решить вопрос
119         ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
                ↪  пересечениями идентификаторов.
120         ///
121         /// Где хранить промежуточный список транзакций?
122         ///
123         /// В оперативной памяти:
124         ///  Минусы:
125         ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
126         ///     так как нужно отдельно выделять память под список трансформаций.
127         ///     2. Выделенной оперативной памяти может не хватить, в том случае,
128         ///     если транзакция использует слишком много трансформаций.
129         ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
130         ///         -> Максимальный размер списка трансформаций можно ограничить / задать
                ↪  константой.
131         ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
                ↪  создавая задержку.
132         ///
133         /// На жёстком диске:
134         ///  Минусы:
135         ///     1. Длительный отклик, на запись каждой трансформации.
136         ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
137         ///         -> Это может решаться упаковкой/исключением дублирующих операций.
138         ///         -> Также это может решаться тем, что короткие транзакции вообще
139         ///            не будут записываться в случае отката.
140         ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
                ↪  операции (трансформации)
141         ///        будут записаны в лог.
142         ///
143         /// </remarks>
144         public class Transaction : DisposableBase
145         {
146             private readonly Queue<Transition> _transitions;
147             private readonly UInt64LinksTransactionsLayer _layer;
148             public bool IsCommitted { get; private set; }
149             public bool IsReverted { get; private set; }
150
151             public Transaction(UInt64LinksTransactionsLayer layer)
152             {
153                 _layer = layer;
154                 if (_layer._currentTransactionId != 0)
155                 {
156                     throw new NotSupportedException("Nested transactions not supported.");
157                 }
158                 IsCommitted = false;
159                 IsReverted = false;
160                 _transitions = new Queue<Transition>();
161                 SetCurrentTransaction(layer, this);
162             }
163
164             public void Commit()
165             {
166                 EnsureTransactionAllowsWriteOperations(this);
167                 while (_transitions.Count > 0)
168                 {
169                     var transition = _transitions.Dequeue();
170                     _layer._transitions.Enqueue(transition);
171                 }
172                 _layer._lastCommitedTransactionId = _layer._currentTransactionId;
173                 IsCommitted = true;
174             }
175
176             private void Revert()
177             {
178                 EnsureTransactionAllowsWriteOperations(this);
179                 var transitionsToRevert = new Transition[_transitions.Count];
180                 _transitions.CopyTo(transitionsToRevert, 0);
181                 for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
182                 {
```

```
                    _layer.RevertTransition(transitionsToRevert[i]);
                }
                IsReverted = true;
            }

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

            public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
            {
                if (transaction.IsReverted)
                {
                    throw new InvalidOperationException("Transation is reverted.");
                }
                if (transaction.IsCommitted)
                {
                    throw new InvalidOperationException("Transation is commited.");
                }
            }

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

```
260             _logAddress = logAddress;
261             _log = FileHelpers.Append(logAddress);
262             _transitions = new Queue<Transition>();
263             _transitionsPusher = new Task(TransitionsPusher);
264             _transitionsPusher.Start();
265         }
266 
267         public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
268 
269         public override ulong Create(IList<ulong> restrictions)
270         {
271             var createdLinkIndex = Links.Create();
272             var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
273             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  default, createdLink));
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         {
```

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

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

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

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

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

        #region DisposalBase

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

        #endregion
    }
}
```

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

```
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 CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<char, TLink>
    {
        private readonly IConverter<TLink> _addressToNumberConverter;
        private readonly TLink _unicodeSymbolMarker;

        public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
        ↪ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
        {
            _addressToNumberConverter = addressToNumberConverter;
```

```
            _unicodeSymbolMarker = unicodeSymbolMarker;
        }

        public TLink Convert(char source)
        {
            var unaryNumber = _addressToNumberConverter.Convert((Integer<TLink>)source);
            return Links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
        }
    }
}
```

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

```
using Platform.Data.Doublets.Sequences.Indexes;
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 StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<string, TLink>
    {
        private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
        private readonly ISequenceIndex<TLink> _index;
        private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
        private readonly TLink _unicodeSequenceMarker;

        public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
        ↪ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
        ↪ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
        {
            _charToUnicodeSymbolConverter = charToUnicodeSymbolConverter;
            _index = index;
            _listToSequenceLinkConverter = listToSequenceLinkConverter;
            _unicodeSequenceMarker = unicodeSequenceMarker;
        }

        public TLink Convert(string source)
        {
            var elements = new TLink[source.Length];
            for (int i = 0; i < source.Length; i++)
            {
                elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
            }
            _index.Add(elements);
            var sequence = _listToSequenceLinkConverter.Convert(elements);
            return Links.GetOrCreate(sequence, _unicodeSequenceMarker);
        }
    }
}
```

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

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

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

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

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

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

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

```
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              {
44                  for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
45                  {
46                      // From NIL to It (NIL -> Character) transformation meaning, (or infinite
                        ↪  amount of NIL characters before actual Character)
47                      var createdLink = _links.CreatePoint();
48                      _links.Update(createdLink, firstLink, createdLink);
49                      if (createdLink != i)
50                      {
51                          throw new InvalidOperationException("Unable to initialize UTF 16
                            ↪  table.");
52                      }
53                  }
54              }
55          }
56
57          // 0 - null link
58          // 1 - nil character (0 character)
59          // ...
60          // 65536 (0(1) + 65535 = 65536 possible values)
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          public static ulong FromCharToLink(char character) => (ulong)character + 1;
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          public static char FromLinkToChar(ulong link) => (char)(link - 1);
67
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          public static bool IsCharLink(ulong link) => link <= MapSize;
70
71          public static string FromLinksToString(IList<ulong> linksList)
72          {
73              var sb = new StringBuilder();
74              for (int i = 0; i < linksList.Count; i++)
75              {
76                  sb.Append(FromLinkToChar(linksList[i]));
77              }
78              return sb.ToString();
79          }
80
81          public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
82          {
83              var sb = new StringBuilder();
84              if (links.Exists(link))
85              {
86                  StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
87                      x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                        ↪  element =>
88                      {
89                          sb.Append(FromLinkToChar(element));
90                          return true;
91                      });
92              }
93              return sb.ToString();
94          }
95
96          public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
            ↪  chars.Length);
97
98          public static ulong[] FromCharsToLinkArray(char[] chars, int count)
99          {
100             // char array to ulong array
101             var linksSequence = new ulong[count];
```

```
            for (var i = 0; i < count; i++)
            {
                linksSequence[i] = FromCharToLink(chars[i]);
            }
            return linksSequence;
        }

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

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

        public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
        {
            var result = new List<ulong[]>();
            var offset = 0;
            while (offset < array.Length)
            {
                var relativeLength = 1;
                if (array[offset] <= LastCharLink)
                {
                    var currentCategory =
                    ↪  CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
                    var absoluteLength = offset + relativeLength;
                    while (absoluteLength < array.Length &&
                           array[absoluteLength] <= LastCharLink &&
                           currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
                           ↪  array[absoluteLength])))
                    {
                        relativeLength++;
                        absoluteLength++;
                    }
                }
                else
                {
                    var absoluteLength = offset + relativeLength;
                    while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
                    {
                        relativeLength++;
                        absoluteLength++;
                    }
                }
                // copy array
                var innerSequence = new ulong[relativeLength];
```

```
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = array[i];
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }
    }
}
```

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

```
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 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

```
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

```
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);
    }
}
```

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

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

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

```
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;
                }
            });

            Func<ulong, ulong, int> compareReference = comparer1.Compare;

            var ts3 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = compareReference(x, y) >= 0;
                }
            });

            var comparer2 = new UInt64Comparer();

            var ts4 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = comparer2.Compare(x, y) >= 0;
                }
            });

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

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

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

namespace Platform.Data.Doublets.Tests
{
    public static class EqualityTests
    {
        protected class UInt64EqualityComparer : IEqualityComparer<ulong>
        {
            public bool Equals(ulong x, ulong y) => x == y;

            public int GetHashCode(ulong obj) => obj.GetHashCode();
        }

        private static bool Equals1<T>(T x, T y) => Equals(x, y);

        private static bool Equals2<T>(T x, T y) => x.Equals(y);

        private static bool Equals3(ulong x, ulong y) => x == y;

        [Fact]
        public static void EqualsPerfomanceTest()
        {
            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 = Equals1(x, y);
                }
            });

            var ts2 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Equals2(x, y);
                }
            });

            var ts3 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Equals3(x, y);
                }
            });

            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

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

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(16)); // Cannot use more because current implementation of tree cuts out 5 bits from the address space.
            Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
            Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
            Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
        }

        private static void Using<TLink>(Action<ILinks<TLink>> action)
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory, ResizableDirectMemoryLinks<TLink>>>())
            {
                action(scope.Use<ILinks<TLink>>());
            }
        }
    }
}
```

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

```
using System;
using System.Linq;
using System.Collections.Generic;
using Xunit;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Doublets.Numbers.Unary;

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

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

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);

                var meaningRoot = links.CreatePoint();
```

```
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
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
```

```
            var readSequence1 = sequences.ToList(optimalVariant);

            Assert.True(sequence.SequenceEqual(readSequence1));
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences;

namespace Platform.Data.Doublets.Tests
{
    public static class ReadSequenceTests
    {
        [Fact]
        public static void ReadSequenceTest()
        {
            const long sequenceLength = 2000;

            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;
                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong> { Walker = new LeveledSequenceWalker<ulong>(links) });

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

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

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

                var sw2 = Stopwatch.StartNew();
                var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();

                var sw3 = Stopwatch.StartNew();
                var readSequence2 = new List<ulong>();
                SequenceWalker.WalkRight(balancedVariant,
                                         links.GetSource,
                                         links.GetTarget,
                                         links.IsPartialPoint,
                                         readSequence2.Add);
                sw3.Stop();

                Assert.True(sequence.SequenceEqual(readSequence1));

                Assert.True(sequence.SequenceEqual(readSequence2));

                // Assert.True(sw2.Elapsed < sw3.Elapsed);

                Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader: {sw2.Elapsed}");

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

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

```
using System.IO;
using Xunit;
using Platform.Singletons;
using Platform.Memory;
using Platform.Data.Doublets.ResizableDirectMemory;
```

```

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

        [Fact]
        public static void BasicFileMappedMemoryTest()
        {
            var tempFilename = Path.GetTempFileName();
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
            File.Delete(tempFilename);
        }

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

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

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

        private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
            var resultLink = _constants.Null;
            memoryAdapter.Each(foundLink =>
            {
                resultLink = foundLink[_constants.IndexPart];
                return _constants.Break;
            }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
            Assert.True(resultLink == link);
            Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
            memoryAdapter.Delete(link);
        }
    }
}
```

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

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

namespace Platform.Data.Doublets.Tests
{
    public static class ScopeTests
    {
        [Fact]
```

```
        public static void SingleDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.IncludeAssemblyOf<IMemory>();
                var instance = scope.Use<IDirectMemory>();
                Assert.IsType<HeapResizableDirectMemory>(instance);
            }
        }

        [Fact]
        public static void CascadeDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.Include<TemporaryFileMappedResizableDirectMemory>();
                scope.Include<UInt64ResizableDirectMemoryLinks>();
                var instance = scope.Use<ILinks<ulong>>();
                Assert.IsType<UInt64ResizableDirectMemoryLinks>(instance);
            }
        }

        [Fact]
        public static void FullAutoResolutionTest()
        {
            using (var scope = new Scope(autoInclude: true, autoExplore: true))
            {
                var instance = scope.Use<UInt64Links>();
                Assert.IsType<UInt64Links>(instance);
            }
        }

        [Fact]
        public static void TypeParametersTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  ResizableDirectMemoryLinks<ulong>>>())
            {
                var links = scope.Use<ILinks<ulong>>();
                Assert.IsType<ResizableDirectMemoryLinks<ulong>>(links);
            }
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Collections;
using Platform.Random;
using Platform.IO;
using Platform.Singletons;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Unicode;

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

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

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

            using (var scope = new TempLinksTestScope(useSequences: true))
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
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
```

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

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

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

                var balancedVariant = balancedVariantConverter.Convert(sequence);

                var partialSequence = new ulong[sequenceLength - 2];

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

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

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

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

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

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

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

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

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

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

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

                var balancedVariant = balancedVariantConverter.Convert(sequence);

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

                var doublet = links.GetSource(balancedVariant);

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

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

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

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

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

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

```

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

                Assert.Contains(doublet, matchedSequences4);
                Assert.Contains(balancedVariant, matchedSequences4);

                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?

```
[![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png ""белая обычная и направленная связи со структурой из 8 цветных элементов последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последовательности"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)

...

[![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif)";

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

        [Fact]
        public static void CompressionTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

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

                var sequence = new[]
                {
                    e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
                };

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

                var compressedVariant = compressingConverter.Convert(sequence);

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

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

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

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

                // 4 - length of sequence
                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0) == sequence[0]);
                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1) == sequence[1]);
                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2) == sequence[2]);
                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3) == sequence[3]);
            }
        }

        [Fact]
        public static void CompressionEfficiencyTest()
```

```
489         {
490             var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
                ↪  StringSplitOptions.RemoveEmptyEntries);
491             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
492             var totalCharacters = arrays.Select(x => x.Length).Sum();
493 
494             using (var scope1 = new TempLinksTestScope(useSequences: true))
495             using (var scope2 = new TempLinksTestScope(useSequences: true))
496             using (var scope3 = new TempLinksTestScope(useSequences: true))
497             {
498                 scope1.Links.Unsync.UseUnicode();
499                 scope2.Links.Unsync.UseUnicode();
500                 scope3.Links.Unsync.UseUnicode();
501 
502                 var balancedVariantConverter1 = new
                    ↪  BalancedVariantConverter<ulong>(scope1.Links.Unsync);
503                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
504                 var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
505                 var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
                    ↪  balancedVariantConverter1, linkFrequenciesCache1,
                    ↪  doInitialFrequenciesIncrement: false);
506 
507                 //var compressor2 = scope2.Sequences;
508                 var compressor3 = scope3.Sequences;
509 
510                 var constants = Default<LinksConstants<ulong>>.Instance;
511 
512                 var sequences = compressor3;
513                 //var meaningRoot = links.CreatePoint();
514                 //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
515                 //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
516                 //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                    ↪  constants.Itself);
517 
518                 //var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
519                 //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
                    ↪  unaryOne);
520                 //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
521                 //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
522                 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
523                 //var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
524 
525                 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
526 
527                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache3);
528 
529                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
                    ↪  linkToItsFrequencyNumberConverter);
530                 var optimalVariantConverter = new
                    ↪  OptimalVariantConverter<ulong>(scope3.Links.Unsync,
                    ↪  sequenceToItsLocalElementLevelsConverter);
531 
532                 var compressed1 = new ulong[arrays.Length];
533                 var compressed2 = new ulong[arrays.Length];
534                 var compressed3 = new ulong[arrays.Length];
535 
536                 var START = 0;
537                 var END = arrays.Length;
538 
539                 //for (int i = START; i < END; i++)
540                 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
541 
542                 var initialCount1 = scope2.Links.Unsync.Count();
543 
544                 var sw1 = Stopwatch.StartNew();
545 
546                 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);
```

```
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();
```

```
682
683                 var START = 0;
684                 var END = arrays.Length;
685
686                 // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
687                 // Stability issue starts at 10001 or 11000
688                 //for (int i = START; i < END; i++)
689                 //{
690                 //    var first = compressor1.Compress(arrays[i]);
691                 //    var second = compressor1.Compress(arrays[i]);
692
693                 //    if (first == second)
694                 //        compressed1[i] = first;
695                 //    else
696                 //    {
697                 //        // TODO: Find a solution for this case
698                 //    }
699                 //}
700
701                 for (int i = START; i < END; i++)
702                 {
703                     var first = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
704                     var second = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
705
706                     if (first == second)
707                     {
708                         compressed1[i] = first;
709                     }
710                     else
711                     {
712                         // TODO: Find a solution for this case
713                     }
714                 }
715
716                 var elapsed1 = sw1.Elapsed;
717
718                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
719
720                 var sw2 = Stopwatch.StartNew();
721
722                 for (int i = START; i < END; i++)
723                 {
724                     var first = balancedVariantConverter.Convert(arrays[i]);
725                     var second = balancedVariantConverter.Convert(arrays[i]);
726
727                     if (first == second)
728                     {
729                         compressed2[i] = first;
730                     }
731                 }
732
733                 var elapsed2 = sw2.Elapsed;
734
735                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    {elapsed2}");
736
737                 Assert.True(elapsed1 > elapsed2);
738
739                 // Checks
740                 for (int i = START; i < END; i++)
741                 {
742                     var sequence1 = compressed1[i];
743                     var sequence2 = compressed2[i];
744
745                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
746                     {
747                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            scope1.Links);
748
749                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            scope2.Links);
750
751                         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                            link.IsPartialPoint());
752                         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                            link.IsPartialPoint());
753
754                         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                            arrays[i].Length > 3)
```

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

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

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

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

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

            //compressor1.ValidateFrequencies();
        }
    }

    [Fact]
    public static void RundomNumbersCompressionQualityTest()
    {
        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]);
            }

```

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

```
904
905                 var sequence = new ulong[sequenceLength];
906                 for (var i = 0; i < sequenceLength; i++)
907                 {
908                     sequence[i] = links.Create();
909                 }
910
911                 var createResults = sequences.CreateAllVariants2(sequence);
912                 var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
913
914                 for (var i = 0; i < 1; i++)
915                 {
916                     var sw1 = Stopwatch.StartNew();
917                     var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
918
919                     var sw2 = Stopwatch.StartNew();
920                     var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
921
922                     var sw3 = Stopwatch.StartNew();
923                     var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
924
925                     var sw4 = Stopwatch.StartNew();
926                     var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
927
928                     Global.Trash = searchResults3;
929                     Global.Trash = searchResults4; //-V3008
930
931                     var intersection1 = createResults.Intersect(searchResults1).ToList();
932                     Assert.True(intersection1.Count == createResults.Length);
933
934                     var intersection2 = reverseResults.Intersect(searchResults1).ToList();
935                     Assert.True(intersection2.Count == reverseResults.Length);
936
937                     var intersection0 = searchResults1.Intersect(searchResults2).ToList();
938                     Assert.True(intersection0.Count == searchResults2.Count);
939
940                     var intersection3 = searchResults2.Intersect(searchResults3).ToList();
941                     Assert.True(intersection3.Count == searchResults3.Count);
942
943                     var intersection4 = searchResults3.Intersect(searchResults4).ToList();
944                     Assert.True(intersection4.Count == searchResults4.Count);
945                 }
946
947                 for (var i = 0; i < sequenceLength; i++)
948                 {
949                     links.Delete(sequence[i]);
950                 }
951             }
952         }
953
954         [Fact(Skip = "Correct implementation is pending")]
955         public static void CalculateAllUsagesTest()
956         {
957             const long sequenceLength = 3;
958
959             using (var scope = new TempLinksTestScope(useSequences: true))
960             {
961                 var links = scope.Links;
962                 var sequences = scope.Sequences;
963
964                 var sequence = new ulong[sequenceLength];
965                 for (var i = 0; i < sequenceLength; i++)
966                 {
967                     sequence[i] = links.Create();
968                 }
969
970                 var createResults = sequences.CreateAllVariants2(sequence);
971
972                 //var reverseResults =
                 ↪  sequences.CreateAllVariants2(sequence.Reverse().ToArray());
973
974                 for (var i = 0; i < 1; i++)
975                 {
976                     var linksTotalUsages1 = new ulong[links.Count() + 1];
977
978                     sequences.CalculateAllUsages(linksTotalUsages1);
979
980                     var linksTotalUsages2 = new ulong[links.Count() + 1];
981
982                     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;
```

```
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);

```

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

            links.Update(linkAddress1, h106E, h108E);

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

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

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

            links.Update(linkAddress2, linkAddress1, h108E);

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

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

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

            links.Update(linkAddress3, linkAddress1, linkAddress2);

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

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

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

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

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

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

            // Update link to reference null (prepare for delete)
            var updated = links.Update(linkAddress3, constants.Null, constants.Null);

            Assert.True(equalityComparer.Equals(updated, linkAddress3));

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

            Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link3.Target, constants.Null));

            // Delete link
            links.Delete(linkAddress3);

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

            var setter3 = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
        }

        public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
        ↪  links, int maximumOperationsPerCycle)
        {
            var comparer = Comparer<TLink>.Default;
            for (var N = 1; N < maximumOperationsPerCycle; N++)
            {
                var random = new System.Random(N);
                var created = 0;
                var deleted = 0;
                for (var i = 0; i < N; i++)
                {
                    long linksCount = (Integer<TLink>)links.Count();
                    var createPoint = random.NextBoolean();
                    if (linksCount > 2 && createPoint)
                    {
                        var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
                        TLink source = (Integer<TLink>)random.NextUInt64(linksAddressRange);
```

```
                        TLink target = (Integer<TLink>)random.NextUInt64(linksAddressRange);
                        ↪ //-V3086
                        var resultLink = links.CreateAndUpdate(source, target);
                        if (comparer.Compare(resultLink, (Integer<TLink>)linksCount) > 0)
                        {
                            created++;
                        }
                    }
                    else
                    {
                        links.Create();
                        created++;
                    }
                }
                Assert.True(created == (Integer<TLink>)links.Count());
                for (var i = 0; i < N; i++)
                {
                    TLink link = (Integer<TLink>)(i + 1);
                    if (links.Exists(link))
                    {
                        links.Delete(link);
                        deleted++;
                    }
                }
                Assert.True((Integer<TLink>)links.Count() == 0);
            }
        }
    }
}
```

./Platform.Data.Doublets.Tests/UInt64LinksTests.cs

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

namespace Platform.Data.Doublets.Tests
{
    public static class UInt64LinksTests
    {
        private static readonly LinksConstants<ulong> _constants =
        ↪ Default<LinksConstants<ulong>>.Instance;

        private const long Iterations = 10 * 1024;

        #region Concept

        [Fact]
        public static void MultipleCreateAndDeleteTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                scope.Links.TestMultipleRandomCreationsAndDeletions(100);
            }
        }

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

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

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

```
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;
```

```
                var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecorator<ulong>)links.Unsync).Links;
                using (var transaction = transactionsLayer.BeginTransaction())
                {
                    var l1 = links.CreateAndUpdate(itself, itself);
                    var l2 = links.CreateAndUpdate(itself, itself);

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

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

                    //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);

                    l2 = links.Update(l2, l1);

                    links.Delete(l2);

                    ExceptionThrower();

                    transaction.Commit();
                }

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

            var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(lastScope.TempTransactionLogFilename);

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

            lastScope.DeleteFiles();
        }
    }

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

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

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

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

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

                links.Unsync.DisposeIfPossible();

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

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

```

```
                    links.Delete(l2);

                    ExceptionThrower();

                    transaction.Commit();
                }

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

            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
            ↪ Scope.TempTransactionLogFilename);

            lastScope.DeleteFiles();
        }
    }

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

        var tempDatabaseFilename = Path.GetTempFileName();
        var tempTransactionLogFilename = Path.GetTempFileName();

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

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

                links.Delete(l1);

                transaction.Commit();
            }

            Global.Trash = links.Count();
        }

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

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

        var tempDatabaseFilename = Path.GetTempFileName();
        var tempTransactionLogFilename = Path.GetTempFileName();

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

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

                links.Delete(l1);

                transaction.Commit();
            }
```

```
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);
```

```
337
338                     ExceptionThrower();
339
340                     transaction.Commit();
341                 }
342
343                 Global.Trash = links.Count();
344             }
345         }
346         catch
347         {
348             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                ↪ TransactionLogFilename);
349         }
350
351         File.Delete(tempDatabaseFilename);
352         File.Delete(tempTransactionLogFilename);
353     }
354
355     private static void ExceptionThrower() => throw new InvalidOperationException();
356
357     [Fact]
358     public static void PathsTest()
359     {
360         var source = _constants.SourcePart;
361         var target = _constants.TargetPart;
362
363         using (var scope = new TempLinksTestScope())
364         {
365             var links = scope.Links;
366             var l1 = links.CreatePoint();
367             var l2 = links.CreatePoint();
368
369             var r1 = links.GetByKeys(l1, source, target, source);
370             var r2 = links.CheckPathExistance(l2, l2, l2, l2);
371         }
372     }
373
374     [Fact]
375     public static void RecursiveStringFormattingTest()
376     {
377         using (var scope = new TempLinksTestScope(useSequences: true))
378         {
379             var links = scope.Links;
380             var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
381
382             var a = links.CreatePoint();
383             var b = links.CreatePoint();
384             var c = links.CreatePoint();
385
386             var ab = links.CreateAndUpdate(a, b);
387             var cb = links.CreateAndUpdate(c, b);
388             var ac = links.CreateAndUpdate(a, c);
389
390             a = links.Update(a, c, b);
391             b = links.Update(b, a, c);
392             c = links.Update(c, a, b);
393
394             Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
395             Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
396             Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
397
398             Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                ↪ "(5:(4:5 (6:5 4)) 6)");
399             Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                ↪ "(6:(5:(4:5 6) 6) 4)");
400             Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
                ↪ "(4:(5:4 (6:5 4)) 6)");
401
402             // 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)"
403
404             Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
                ↪ "{{5}{5}{4}{6}}");
405             Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                ↪ "{{5}{6}{6}{4}}");
406             Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                ↪ "{{4}{5}{4}{6}}");
407         }
```

```
408         }
409
410         private static void DefaultFormatter(StringBuilder sb, ulong link)
411         {
412             sb.Append(link.ToString());
413         }
414
415         #endregion
416
417         #region Performance
418
419         /*
420         public static void RunAllPerformanceTests()
421         {
422             try
423             {
424                 links.TestLinksInSteps();
425             }
426             catch (Exception ex)
427             {
428                 ex.WriteToConsole();
429             }
430
431             return;
432
433             try
434             {
435                 //ThreadPool.SetMaxThreads(2, 2);
436
437                 // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    ↪ результат
438                 // Также это дополнительно помогает в отладке
439                 // Увеличивает вероятность попадания информации в кэши
440                 for (var i = 0; i < 10; i++)
441                 {
442                     //0 - 10 ГБ
443                     //Каждые 100 МБ срез цифр
444
445                     //links.TestGetSourceFunction();
446                     //links.TestGetSourceFunctionInParallel();
447                     //links.TestGetTargetFunction();
448                     //links.TestGetTargetFunctionInParallel();
449                     links.Create64BillionLinks();
450
451                     links.TestRandomSearchFixed();
452                     //links.Create64BillionLinksInParallel();
453                     links.TestEachFunction();
454                     //links.TestForeach();
455                     //links.TestParallelForeach();
456                 }
457
458                 links.TestDeletionOfAllLinks();
459
460             }
461             catch (Exception ex)
462             {
463                 ex.WriteToConsole();
464             }
465         }*/
466
467         /*
468         public static void TestLinksInSteps()
469         {
470             const long gibibyte = 1024 * 1024 * 1024;
471             const long mebibyte = 1024 * 1024;
472
473             var totalLinksToCreate = gibibyte /
    ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
474             var linksStep = 102 * mebibyte /
    ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
475
476             var creationMeasurements = new List<TimeSpan>();
477             var searchMeasuremets = new List<TimeSpan>();
478             var deletionMeasurements = new List<TimeSpan>();
479
480             GetBaseRandomLoopOverhead(linksStep);
481             GetBaseRandomLoopOverhead(linksStep);
482
483             var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
484
```

```
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
```

```
559                 //var firstLink = links.First();
560                 // Создаём одну связь, из которой будет производить считывание
561                 var firstLink = links.Create();
562 
563                 var sw = Stopwatch.StartNew();
564 
565                 // Тестируем саму функцию
566                 for (ulong i = 0; i < Iterations; i++)
567                 {
568                     counter += links.GetSource(firstLink);
569                 }
570 
571                 var elapsedTime = sw.Elapsed;
572 
573                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
574 
575                 // Удаляем связь, из которой производилось считывание
576                 links.Delete(firstLink);
577 
578                 ConsoleHelpers.Debug(
579                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
580                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
581             }
582         }
583 
584         [Fact(Skip = "performance test")]
585         public static void GetSourceInParallel()
586         {
587             using (var scope = new TempLinksTestScope())
588             {
589                 var links = scope.Links;
590                 ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
                    ↪  parallel.", Iterations);
591 
592                 long counter = 0;
593 
594                 //var firstLink = links.First();
595                 var firstLink = links.Create();
596 
597                 var sw = Stopwatch.StartNew();
598 
599                 // Тестируем саму функцию
600                 Parallel.For(0, Iterations, x =>
601                 {
602                     Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
603                     //Interlocked.Increment(ref counter);
604                 });
605 
606                 var elapsedTime = sw.Elapsed;
607 
608                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
609 
610                 links.Delete(firstLink);
611 
612                 ConsoleHelpers.Debug(
613                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
614                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
615             }
616         }
617 
618         [Fact(Skip = "performance test")]
619         public static void TestGetTarget()
620         {
621             using (var scope = new TempLinksTestScope())
622             {
623                 var links = scope.Links;
624                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
                    ↪  Iterations);
625 
626                 ulong counter = 0;
627 
628                 //var firstLink = links.First();
629                 var firstLink = links.Create();
630 
631                 var sw = Stopwatch.StartNew();
632 
633                 for (ulong i = 0; i < Iterations; i++)
```

```
                {
                    counter += links.GetTarget(firstLink);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

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

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

                long counter = 0;

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

                var sw = Stopwatch.StartNew();

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

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

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

        // TODO: Заполнить базу данных перед тестом
        /*
        [Fact]
        public void TestRandomSearchFixed()
        {
            var tempFilename = Path.GetTempFileName();

            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
            {
                long iterations = 64 * 1024 * 1024 /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;

                ulong counter = 0;
                var maxLink = links.Total;

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

                var sw = Stopwatch.StartNew();

                for (var i = iterations; i > 0; i--)
                {
                    var source =
    ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
                    var target =
    ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);

```

```
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: 0(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: 0(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                     {
```

```
                        links.Create();
                    }
                });

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

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

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

        [Fact(Skip = "performance test")]
        public static void Create64BillionLinksInParallel()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var linksBeforeTest = links.Count();

                var sw = Stopwatch.StartNew();

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

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

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

                var elapsedTime = sw.Elapsed;

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

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

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

                ConsoleHelpers.Debug("Deleting all links");

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

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

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

        #endregion
    }
}
```

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

```
using Xunit;
using Platform.Random;
using Platform.Data.Doublets.Numbers.Unary;

namespace Platform.Data.Doublets.Tests
{
    public static class UnaryNumberConvertersTests
    {
        [Fact]
        public static void ConvertersTest()
        {
```

```
            using (var scope = new TempLinksTestScope())
            {
                const int N = 10;
                var links = scope.Links;
                var meaningRoot = links.CreatePoint();
                var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
                var powerOf2ToUnaryNumberConverter = new
                ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
                var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                var random = new System.Random(0);
                ulong[] numbers = new ulong[N];
                ulong[] unaryNumbers = new ulong[N];
                for (int i = 0; i < N; i++)
                {
                    numbers[i] = random.NextUInt64();
                    unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
                }
                var fromUnaryNumberConverterUsingOrOperation = new
                ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                var fromUnaryNumberConverterUsingAddOperation = new
                ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
                for (int i = 0; i < N; i++)
                {
                    Assert.Equal(numbers[i],
                    ↪  fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
                    Assert.Equal(numbers[i],
                    ↪  fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
                }
            }
        }
    }
}
```

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

```
using Xunit;
using Platform.Interfaces;
using Platform.Memory;
using Platform.Reflection;
using Platform.Scopes;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Numbers.Raw;
using Platform.Data.Doublets.Numbers.Unary;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.ResizableDirectMemory;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Unicode;

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

        [Fact]
        public static void CharAndRawNumberUnicodeSymbolConvertersTest()
        {
```

```
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";
```

```
92
93                  var unicodeSequenceLink =
                    ↪  stringToUnicodeSequenceConverter.Convert(originalString);
94
95                  var unicodeSymbolCriterionMatcher = new
                    ↪  UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
96                  var unicodeSymbolToCharConverter = new
                    ↪  UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
                    ↪  unicodeSymbolCriterionMatcher);
97
98                  var unicodeSequenceCriterionMatcher = new
                    ↪  UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
99
100                 var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
                    ↪  unicodeSymbolCriterionMatcher.IsMatched);
101
102                 var unicodeSequenceToStringConverter = new
                    ↪  UnicodeSequenceToStringConverter<ulong>(links,
                    ↪  unicodeSequenceCriterionMatcher, sequenceWalker,
                    ↪  unicodeSymbolToCharConverter);
103
104                 var resultingString =
                    ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
105
106                 Assert.Equal(originalString, resultingString);
107             }
108         }
109     }
110 }
```

Index

./Platform.Data.Doublets.Tests/ComparisonTests.cs, 142
./Platform.Data.Doublets.Tests/EqualityTests.cs, 143
./Platform.Data.Doublets.Tests/GenericLinksTests.cs, 144
./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 145
./Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 147
./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 147
./Platform.Data.Doublets.Tests/ScopeTests.cs, 148
./Platform.Data.Doublets.Tests/SequencesTests.cs, 149
./Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 164
./Platform.Data.Doublets.Tests/TestExtensions.cs, 164
./Platform.Data.Doublets.Tests/UInt64LinksTests.cs, 167
./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 179
./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 180
./Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs, 1
./Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs, 1
./Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs, 1
./Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs, 2
./Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs, 2
./Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs, 3
./Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs, 3
./Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs, 4
./Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs, 4
./Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs, 4
./Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs, 5
./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs, 5
./Platform.Data.Doublets/Decorators/UInt64Links.cs, 5
./Platform.Data.Doublets/Decorators/UniLinks.cs, 7
./Platform.Data.Doublets/Doublet.cs, 11
./Platform.Data.Doublets/DoubletComparer.cs, 11
./Platform.Data.Doublets/Hybrid.cs, 12
./Platform.Data.Doublets/ILinks.cs, 14
./Platform.Data.Doublets/ILinksExtensions.cs, 14
./Platform.Data.Doublets/ISynchronizedLinks.cs, 26
./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs, 25
./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs, 25
./Platform.Data.Doublets/Link.cs, 26
./Platform.Data.Doublets/LinkExtensions.cs, 28
./Platform.Data.Doublets/LinksOperatorBase.cs, 28
./Platform.Data.Doublets/Numbers/Raw/AddressToRawNumberConverter.cs, 29
./Platform.Data.Doublets/Numbers/Raw/RawNumberToAddressConverter.cs, 29
./Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 29
./Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConveter.cs, 29
./Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 30
./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 31
./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 31
./Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 32
./Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 33
./Platform.Data.Doublets/ResizableDirectMemory/ILinksListMethods.cs, 34
./Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.cs, 34
./Platform.Data.Doublets/ResizableDirectMemory/LinksAVLBalancedTreeMethodsBase.cs, 34
./Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs, 38
./Platform.Data.Doublets/ResizableDirectMemory/LinksSourcesAVLBalancedTreeMethods.cs, 38
./Platform.Data.Doublets/ResizableDirectMemory/LinksTargetsAVLBalancedTreeMethods.cs, 39
./Platform.Data.Doublets/ResizableDirectMemory/RawLink.cs, 41
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 41
./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksAVLBalancedTreeMethodsBase.cs, 48
./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksHeader.cs, 51
./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksSourcesAVLBalancedTreeMethods.cs, 52
./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksTargetsAVLBalancedTreeMethods.cs, 53
./Platform.Data.Doublets/ResizableDirectMemory/UInt64RawLink.cs, 54
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 54
./Platform.Data.Doublets/ResizableDirectMemory/UInt64UnusedLinksListMethods.cs, 61
./Platform.Data.Doublets/ResizableDirectMemory/UnusedLinksListMethods.cs, 62
./Platform.Data.Doublets/Sequences/ArrayExtensions.cs, 62
./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 63
./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 63

./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 66
./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 67
./Platform.Data.Doublets/Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs, 68
./Platform.Data.Doublets/Sequences/CreteriaMatchers/DefaultSequenceElementCriterionMatcher.cs, 69
./Platform.Data.Doublets/Sequences/CreteriaMatchers/MarkedSequenceCriterionMatcher.cs, 69
./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 69
./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 70
./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 70
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 72
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 74
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.cs, 75
./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 75
./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 75
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 76
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 76
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 77
./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 78
./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 78
./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 79
./Platform.Data.Doublets/Sequences/IListExtensions.cs, 79
./Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 79
./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 80
./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 81
./Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 81
./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 82
./Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 82
./Platform.Data.Doublets/Sequences/ListFiller.cs, 82
./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 93
./Platform.Data.Doublets/Sequences/Sequences.cs, 83
./Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 119
./Platform.Data.Doublets/Sequences/SequencesOptions.cs, 120
./Platform.Data.Doublets/Sequences/SetFiller.cs, 121
./Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 122
./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 122
./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 123
./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 124
./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 125
./Platform.Data.Doublets/Stacks/Stack.cs, 126
./Platform.Data.Doublets/Stacks/StackExtensions.cs, 126
./Platform.Data.Doublets/SynchronizedLinks.cs, 127
./Platform.Data.Doublets/UInt64Link.cs, 127
./Platform.Data.Doublets/UInt64LinkExtensions.cs, 130
./Platform.Data.Doublets/UInt64LinksExtensions.cs, 130
./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 132
./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 137
./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 138
./Platform.Data.Doublets/Unicode/UnicodeMap.cs, 138
./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 142