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

```csharp
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Data.Doublets.Decorators
6  {
7      public class LinksCascadeUniquenessAndUsagesResolver<TLink> : LinksUniquenessResolver<TLink>
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLink> links) : base(links) { }
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
           ↪ newLinkAddress)
14         {
15             // Use Facade (the last decorator) to ensure recursion working correctly
16             Facade.MergeUsages(oldLinkAddress, newLinkAddress);
17             return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
18         }
19     }
20 }
```

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

```csharp
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      /// <remarks>
9      /// <para>Must be used in conjunction with NonNullContentsLinkDeletionResolver.</para>
10     /// <para>Должен использоваться вместе с NonNullContentsLinkDeletionResolver.</para>
11     /// </remarks>
12     public class LinksCascadeUsagesResolver<TLink> : LinksDecoratorBase<TLink>
13     {
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public LinksCascadeUsagesResolver(ILinks<TLink> links) : base(links) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public override void Delete(IList<TLink> restrictions)
19         {
20             var linkIndex = restrictions[Constants.IndexPart];
21             // Use Facade (the last decorator) to ensure recursion working correctly
22             Facade.DeleteAllUsages(linkIndex);
23             Links.Delete(linkIndex);
24         }
25     }
26 }
```

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

```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
10     {
11         private ILinks<TLink> _facade;
12
13         public LinksConstants<TLink> Constants { get; }
14
15         public ILinks<TLink> Facade
16         {
17             get => _facade;
18             set
19             {
20                 _facade = value;
21                 if (Links is LinksDecoratorBase<TLink> decorator)
22                 {
23                     decorator.Facade = value;
24                 }
25                 else if (Links is LinksDisposableDecoratorBase<TLink> disposableDecorator)
26                 {
```

```csharp
                       disposableDecorator.Facade = value;
                   }
               }
           }
       }

       [MethodImpl(MethodImplOptions.AggressiveInlining)]
       protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
       {
           Constants = links.Constants;
           Facade = this;
       }

       [MethodImpl(MethodImplOptions.AggressiveInlining)]
       public virtual TLink Count(IList<TLink> restrictions) => Links.Count(restrictions);

       [MethodImpl(MethodImplOptions.AggressiveInlining)]
       public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
       ↪   => Links.Each(handler, restrictions);

       [MethodImpl(MethodImplOptions.AggressiveInlining)]
       public virtual TLink Create(IList<TLink> restrictions) => Links.Create(restrictions);

       [MethodImpl(MethodImplOptions.AggressiveInlining)]
       public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
       ↪   Links.Update(restrictions, substitution);

       [MethodImpl(MethodImplOptions.AggressiveInlining)]
       public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);
   }
}
```

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

```csharp
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
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>
    {
        private ILinks<TLink> _facade;

        public LinksConstants<TLink> Constants { get; }

        public ILinks<TLink> Links { get; }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksDisposableDecoratorBase(ILinks<TLink> links)
        {
            Links = links;
            Constants = links.Constants;
            Facade = this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Count(IList<TLink> restrictions) => Links.Count(restrictions);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        ↪   => Links.Each(handler, restrictions);
```

```
48
49            [MethodImpl(MethodImplOptions.AggressiveInlining)]
50            public virtual TLink Create(IList<TLink> restrictions) => Links.Create(restrictions);
51
52            [MethodImpl(MethodImplOptions.AggressiveInlining)]
53            public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
   ↪        Links.Update(restrictions, substitution);
54
55            [MethodImpl(MethodImplOptions.AggressiveInlining)]
56            public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);
57
58            protected override bool AllowMultipleDisposeCalls => true;
59
60            protected override void Dispose(bool manual, bool wasDisposed)
61            {
62                if (!wasDisposed)
63                {
64                    Links.DisposeIfPossible();
65                }
66            }
67        }
68    }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Decorators
8   {
9       // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
   ↪    be external (hybrid link's raw number).
10      public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
11      {
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
17          {
18              Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
19              return Links.Each(handler, restrictions);
20          }
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
24          {
25              // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
26              Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
27              Links.EnsureInnerReferenceExists(substitution, nameof(substitution));
28              return Links.Update(restrictions, substitution);
29          }
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public override void Delete(IList<TLink> restrictions)
33          {
34              var link = restrictions[Constants.IndexPart];
35              Links.EnsureLinkExists(link, nameof(link));
36              Links.Delete(link);
37          }
38      }
39  }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Decorators
8   {
9       public class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
10      {
11          private static readonly EqualityComparer<TLink> _equalityComparer =
   ↪        EqualityComparer<TLink>.Default;
12
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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

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

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

namespace Platform.Data.Doublets.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>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksNonExistentDependenciesCreator(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Create(IList<TLink> restrictions)
```

```
15            {
16                var link = Links.Create();
17                return Links.Update(link, link, link);
18            }
19
20            [MethodImpl(MethodImplOptions.AggressiveInlining)]
21            public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
   ↳      Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Null,
   ↳      restrictions, substitution));
22        }
23    }
```

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

```
1    using System.Collections.Generic;
2    using System.Runtime.CompilerServices;
3
4    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6    namespace Platform.Data.Doublets.Decorators
7    {
8        public class LinksUniquenessResolver<TLink> : LinksDecoratorBase<TLink>
9        {
10            private static readonly EqualityComparer<TLink> _equalityComparer =
    ↳      EqualityComparer<TLink>.Default;
11
12            [MethodImpl(MethodImplOptions.AggressiveInlining)]
13            public LinksUniquenessResolver(ILinks<TLink> links) : base(links) { }
14
15            [MethodImpl(MethodImplOptions.AggressiveInlining)]
16            public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
17            {
18                var newLinkAddress = Links.SearchOrDefault(substitution[Constants.SourcePart],
    ↳      substitution[Constants.TargetPart]);
19                if (_equalityComparer.Equals(newLinkAddress, default))
20                {
21                    return Links.Update(restrictions, substitution);
22                }
23                return ResolveAddressChangeConflict(restrictions[Constants.IndexPart],
    ↳      newLinkAddress);
24            }
25
26            [MethodImpl(MethodImplOptions.AggressiveInlining)]
27            protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
    ↳      newLinkAddress)
28            {
29                if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
    ↳      Links.Exists(oldLinkAddress))
30                {
31                    Facade.Delete(oldLinkAddress);
32                }
33                return newLinkAddress;
34            }
35        }
36    }
```

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

```
1    using System.Collections.Generic;
2    using System.Runtime.CompilerServices;
3
4    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6    namespace Platform.Data.Doublets.Decorators
7    {
8        public class LinksUniquenessValidator<TLink> : LinksDecoratorBase<TLink>
9        {
10            [MethodImpl(MethodImplOptions.AggressiveInlining)]
11            public LinksUniquenessValidator(ILinks<TLink> links) : base(links) { }
12
13            [MethodImpl(MethodImplOptions.AggressiveInlining)]
14            public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
15            {
16                Links.EnsureDoesNotExists(substitution[Constants.SourcePart],
    ↳      substitution[Constants.TargetPart]);
17                return Links.Update(restrictions, substitution);
18            }
19        }
20    }
```

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

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Decorators
7   {
8       public class LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public LinksUsagesValidator(ILinks<TLink> links) : base(links) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
15          {
16              Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
17              return Links.Update(restrictions, substitution);
18          }
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          public override void Delete(IList<TLink> restrictions)
22          {
23              var link = restrictions[Constants.IndexPart];
24              Links.EnsureNoUsages(link);
25              Links.Delete(link);
26          }
27      }
28  }
```

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

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Decorators
7   {
8       public class NonNullContentsLinkDeletionResolver<TLink> : LinksDecoratorBase<TLink>
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public NonNullContentsLinkDeletionResolver(ILinks<TLink> links) : base(links) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public override void Delete(IList<TLink> restrictions)
15          {
16              var linkIndex = restrictions[Constants.IndexPart];
17              Links.EnforceResetValues(linkIndex);
18              Links.Delete(linkIndex);
19          }
20      }
21  }
```

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

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Decorators
7   {
8       /// <summary>
9       /// Представляет объект для работы с базой данных (файлом) в формате Links (массива связей).
10      /// </summary>
11      /// <remarks>
12      /// Возможные оптимизации:
13      /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
14      ///     + меньше объём БД
15      ///     - меньше производительность
16      ///     - больше ограничение на количество связей в БД)
17      /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
18      ///     + меньше объём БД
19      ///     - больше сложность
20      ///
21      /// 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
22      /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
        ↪    (битовыми строками) - вариант матрицы (выстраеваемой лениво).
```

```
23        ///
24        /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
          ↪    выбрасываться только при #if DEBUG
25        /// </remarks>
26        public class UInt64Links : LinksDisposableDecoratorBase<ulong>
27        {
28            [MethodImpl(MethodImplOptions.AggressiveInlining)]
29            public UInt64Links(ILinks<ulong> links) : base(links) { }
30
31            [MethodImpl(MethodImplOptions.AggressiveInlining)]
32            public override ulong Create(IList<ulong> restrictions) => Links.CreatePoint();
33
34            public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
35            {
36                var constants = Constants;
37                var indexPartConstant = constants.IndexPart;
38                var updatedLink = restrictions[indexPartConstant];
39                var sourcePartConstant = constants.SourcePart;
40                var newSource = substitution[sourcePartConstant];
41                var targetPartConstant = constants.TargetPart;
42                var newTarget = substitution[targetPartConstant];
43                var nullConstant = constants.Null;
44                var existedLink = nullConstant;
45                var itselfConstant = constants.Itself;
46                if (newSource != itselfConstant && newTarget != itselfConstant)
47                {
48                    existedLink = Links.SearchOrDefault(newSource, newTarget);
49                }
50                if (existedLink == nullConstant)
51                {
52                    var before = Links.GetLink(updatedLink);
53                    if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
                      ↪    newTarget)
54                    {
55                        Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
                          ↪    newSource,
56                                                  newTarget == itselfConstant ? updatedLink :
                                                  ↪    newTarget);
57                    }
58                    return updatedLink;
59                }
60                else
61                {
62                    return Facade.MergeAndDelete(updatedLink, existedLink);
63                }
64            }
65
66            [MethodImpl(MethodImplOptions.AggressiveInlining)]
67            public override void Delete(IList<ulong> restrictions)
68            {
69                var linkIndex = restrictions[Constants.IndexPart];
70                Links.EnforceResetValues(linkIndex);
71                Facade.DeleteAllUsages(linkIndex);
72                Links.Delete(linkIndex);
73            }
74        }
75    }
```

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

```
1    using System;
2    using System.Collections.Generic;
3    using System.Linq;
4    using Platform.Collections;
5    using Platform.Collections.Arrays;
6    using Platform.Collections.Lists;
7    using Platform.Data.Universal;
8
9    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11   namespace Platform.Data.Doublets.Decorators
12   {
13       /// <remarks>
14       /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
15       /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed
           ↪    by itself. But can cause creation (update from nothing) or deletion (update to nothing).
16       ///
17       /// 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.)
```

```csharp
        /// </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))
                ////                    transitions.Add(new Transition(matchedLink, newValue));
                ////            }
                ////            else
                ////            {
                ////                Func<T, bool> handler;
                ////                handler = link =>
                ////                {
                ////                    var matchedLink = Memory.GetLinkValue(link);
                ////                    var newValue = Memory.GetLinkValue(link);
                ////                    newValue[Constants.IndexPart] = Constants.Itself;
                ////                    newValue[Constants.SourcePart] =
                ////    Equals(substitution[Constants.SourcePart], Constants.Itself) ?
                ////    matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
                ////                    newValue[Constants.TargetPart] =
                ////    Equals(substitution[Constants.TargetPart], Constants.Itself) ?
                ////    matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
                ////                    var matchDecision = matchedHandler(matchedLink, newValue);
                ////                    if (Equals(matchDecision, Constants.Break))
                ////                        return false;
                ////                    if (!Equals(matchDecision, Constants.Skip))
                ////                        transitions.Add(new Transition(matchedLink, newValue));
                ////                    return true;
                ////                };
                ////                if (!Memory.Each(handler, restriction))
                ////                    return Constants.Break;
                ////            }
```

```
 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;
```

```csharp
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                 }
```

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

        /// <remarks>
```

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

./Platform.Data.Doublets/DoubletComparer.cs

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

./Platform.Data.Doublets/Doublet.cs

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

```
26
27          public override int GetHashCode() => (Source, Target).GetHashCode();
28      }
29  }
```

./Platform.Data.Doublets/Hybrid.cs

```
1  using System;
2  using System.Reflection;
3  using System.Reflection.Emit;
4  using Platform.Reflection;
5  using Platform.Converters;
6  using Platform.Exceptions;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets
11 {
12     public struct Hybrid<T>
13     {
14         private static readonly Func<object, T> _absAndConvert;
15         private static readonly Func<object, T> _absAndNegateAndConvert;
16
17         static Hybrid()
18         {
19             _absAndConvert = DelegateHelpers.Compile<Func<object, T>>(emiter =>
20             {
21                 Ensure.Always.IsUnsignedInteger<T>();
22                 emiter.LoadArgument(0);
23                 var signedVersion = NumericType<T>.SignedVersion;
24                 var signedVersionField =
25                 ↪   typeof(NumericType<T>).GetTypeInfo().GetField("SignedVersion",
26                 ↪   BindingFlags.Static | BindingFlags.Public);
27                 //emiter.LoadField(signedVersionField);
28                 emiter.Emit(OpCodes.Ldsfld, signedVersionField);
29                 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)));
```

```csharp
        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);

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

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

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

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

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

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

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

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

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

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

./Platform.Data.Doublets/ILinks.cs

```csharp
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

```csharp
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,
```

```csharp
        /// чтобы удалился весь контент)
        /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
        /// </remarks>
        public static void DeleteAll<TLink>(this ILinks<TLink> links)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            var comparer = Comparer<TLink>.Default;
            for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
            ↪ Arithmetic.Decrement(i))
            {
                links.Delete(i);
                if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
                {
                    i = links.Count();
                }
            }
        }

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

        #region Paths

        /// <remarks>
        /// TODO: Как так? Как то что ниже может быть корректно?
        /// Скорее всего практически не применимо
        /// Предполагалось, что можно было конвертировать формируемый в проходе через
        ↪ SequenceWalker
        /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
        /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
        /// </remarks>
        public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
        ↪ path)
        {
            var current = path[0];
            //EnsureLinkExists(current, "path");
            if (!links.Exists(current))
            {
                return false;
            }
            var equalityComparer = EqualityComparer<TLink>.Default;
            var constants = links.Constants;
            for (var i = 1; i < path.Length; i++)
            {
                var next = path[i];
                var values = links.GetLink(current);
                var source = values[constants.SourcePart];
                var target = values[constants.TargetPart];
                if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                ↪ next))
                {
                    //throw new InvalidOperationException(string.Format("Невозможно выбрать
                    ↪ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
                    return false;
                }
                if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                ↪ target))
                {
                    //throw new InvalidOperationException(string.Format("Невозможно продолжить
                    ↪ путь через элемент пути {0}", next));
```

```csharp
                    return false;
                }
                current = next;
            }
            return true;
        }

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

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

        #endregion

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

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

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

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

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

        /// <summary>
        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
        ///     (handler) для каждой подходящей связи.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Значение, определяющее соответствующие шаблону связи.
        ///     (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
        ///     Constants.Any - любое начало, 1..∞ конкретное начало)</param>
        /// <param name="target">Значение, определяющее соответствующие шаблону связи.
        ///     (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
        ///     Constants.Any - любой конец, 1..∞ конкретный конец)</param>
        /// <param name="handler">Обработчик каждой подходящей связи.</param>
        /// <returns>True, в случае если проход по связям не был прерван и False в обратном
        ///     случае.</returns>
```

```csharp
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 EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↪   restrictions)
302        {
303            for (var i = 0; i < restrictions.Count; i++)
304            {
305                if (!links.Exists(restrictions[i]))
306                {
307                    throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
    ↪   $"sequence[{i}]");
308                }
309            }
310        }
311
312        [MethodImpl(MethodImplOptions.AggressiveInlining)]
313        public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
    ↪   reference, string argumentName)
314        {
315            if (links.Constants.IsInnerReference(reference) && !links.Exists(reference))
316            {
317                throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
318            }
319        }
320
321        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
        ↪  IList<TLink> restrictions, string argumentName)
        {
            for (int i = 0; i < restrictions.Count; i++)
            {
                links.EnsureInnerReferenceExists(restrictions[i], argumentName);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
        ↪  restrictions)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            var any = links.Constants.Any;
            for (var i = 0; i < restrictions.Count; i++)
            {
                if (!equalityComparer.Equals(restrictions[i], any) &&
                ↪  !links.Exists(restrictions[i]))
                {
                    throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                    ↪  $"sequence[{i}]");
                }
            }
        }

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

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

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

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

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

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

        /// <param name="links">Хранилище связей.</param>
```

```csharp
390        public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
         ↪  params TLink[] addresses)
391        {
392            var constants = links.Constants;
393
394            var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
             ↪  !links.Exists(x)));
395            if (nonExistentAddresses.Count > 0)
396            {
397                var max = nonExistentAddresses.Max();
398                max = (Integer<TLink>)System.Math.Min((ulong)(Integer<TLink>)max,
                 ↪  (ulong)(Integer<TLink>)constants.PossibleInnerReferencesRange.Maximum);
399                var createdLinks = new List<TLink>();
400                var equalityComparer = EqualityComparer<TLink>.Default;
401                TLink createdLink = creator();
402                while (!equalityComparer.Equals(createdLink, max))
403                {
404                    createdLinks.Add(createdLink);
405                }
406                for (var i = 0; i < createdLinks.Count; i++)
407                {
408                    if (!nonExistentAddresses.Contains(createdLinks[i]))
409                    {
410                        links.Delete(createdLinks[i]);
411                    }
412                }
413            }
414        }
415
416        #endregion
417
418        /// <param name="links">Хранилище связей.</param>
419        public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
420        {
421            var constants = links.Constants;
422            var values = links.GetLink(link);
423            TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
             ↪  constants.Any));
424            var equalityComparer = EqualityComparer<TLink>.Default;
425            if (equalityComparer.Equals(values[constants.SourcePart], link))
426            {
427                usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
428            }
429            TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
             ↪  link));
430            if (equalityComparer.Equals(values[constants.TargetPart], link))
431            {
432                usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
433            }
434            return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
435        }
436
437        /// <param name="links">Хранилище связей.</param>
438        [MethodImpl(MethodImplOptions.AggressiveInlining)]
439        public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
         ↪  Comparer<TLink>.Default.Compare(links.CountUsages(link), Integer<TLink>.Zero) > 0;
440
441        /// <param name="links">Хранилище связей.</param>
442        [MethodImpl(MethodImplOptions.AggressiveInlining)]
443        public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
         ↪  TLink target)
444        {
445            var constants = links.Constants;
446            var values = links.GetLink(link);
447            var equalityComparer = EqualityComparer<TLink>.Default;
448            return equalityComparer.Equals(values[constants.SourcePart], source) &&
             ↪  equalityComparer.Equals(values[constants.TargetPart], target);
449        }
450
451        /// <summary>
452        /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
453        /// </summary>
454        /// <param name="links">Хранилище связей.</param>
455        /// <param name="source">Индекс связи, которая является началом для искомой
         ↪  связи.</param>
456        /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
457        /// <returns>Индекс искомой связи с указанными Source (началом) и Target
         ↪  (концом).</returns>
```

```csharp
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
                ↪   target)
                {
                    var contants = links.Constants;
                    var setter = new Setter<TLink, TLink>(contants.Continue, contants.Break, default);
                    links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
                    return setter.Result;
                }

                /// <param name="links">Хранилище связей.</param>
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);

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

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

                /// <summary>
                /// Обновляет связь с указанными началом (Source) и концом (Target)
                /// на связь с указанными началом (NewSource) и концом (NewTarget).
                /// </summary>
                /// <param name="links">Хранилище связей.</param>
                /// <param name="link">Индекс обновляемой связи.</param>
                /// <param name="newSource">Индекс связи, которая является началом связи, на которую
                ↪   выполняется обновление.</param>
                /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
                ↪   выполняется обновление.</param>
                /// <returns>Индекс обновлённой связи.</returns>
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
                ↪   TLink newTarget) => links.Update(new LinkAddress<TLink>(link), new Link<TLink>(link,
                ↪   newSource, newTarget));

                /// <summary>
                /// Обновляет связь с указанными началом (Source) и концом (Target)
                /// на связь с указанными началом (NewSource) и концом (NewTarget).
                /// </summary>
                /// <param name="links">Хранилище связей.</param>
                /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
                ↪   может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
                ↪   Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
                ↪   связи.</param>
                /// <returns>Индекс обновлённой связи.</returns>
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
                {
                    if (restrictions.Length == 2)
                    {
                        return links.MergeAndDelete(restrictions[0], restrictions[1]);
                    }
                    if (restrictions.Length == 4)
                    {
                        return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
                        ↪   restrictions[2], restrictions[3]);
                    }
                    else
                    {
                        return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
                    }
                }

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
                ↪   links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
                {
                    var equalityComparer = EqualityComparer<TLink>.Default;
                    var constants = links.Constants;
                    var restrictionsIndex = restrictions[constants.IndexPart];
```

```csharp
                var substitutionIndex = substitution[constants.IndexPart];
                if (equalityComparer.Equals(substitutionIndex, default))
                {
                    substitutionIndex = restrictionsIndex;
                }
                var source = substitution[constants.SourcePart];
                var target = substitution[constants.TargetPart];
                source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
                target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
                return new Link<TLink>(substitutionIndex, source, target);
            }

            /// <summary>
            /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
            ↪  с указанными Source (началом) и Target (концом).
            /// </summary>
            /// <param name="links">Хранилище связей.</param>
            /// <param name="source">Индекс связи, которая является началом на создаваемой
            ↪  связи.</param>
            /// <param name="target">Индекс связи, которая является концом для создаваемой
            ↪  связи.</param>
            /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪  target)
            {
                var link = links.SearchOrDefault(source, target);
                if (EqualityComparer<TLink>.Default.Equals(link, default))
                {
                    link = links.CreateAndUpdate(source, target);
                }
                return link;
            }

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

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

```csharp
592                    return link;
593                }
594                return default;
595            }
596
597            /// <summary>Удаляет несколько связей.</summary>
598            /// <param name="links">Хранилище связей.</param>
599            /// <param name="deletedLinks">Список адресов связей к удалению.</param>
600            [MethodImpl(MethodImplOptions.AggressiveInlining)]
601            public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
602            {
603                for (int i = 0; i < deletedLinks.Count; i++)
604                {
605                    links.Delete(deletedLinks[i]);
606                }
607            }
608
609            /// <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>
610            public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
611            {
612                var anyConstant = links.Constants.Any;
613                var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
614                links.DeleteByQuery(usagesAsSourceQuery);
615                var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
616                links.DeleteByQuery(usagesAsTargetQuery);
617            }
618
619            public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
620            {
621                var count = (Integer<TLink>)links.Count(query);
622                if (count > 0)
623                {
624                    var queryResult = new TLink[count];
625                    var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
                        ↪  links.Constants.Continue);
626                    links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
627                    for (var i = (long)count - 1; i >= 0; i--)
628                    {
629                        links.Delete(queryResult[i]);
630                    }
631                }
632            }
633
634            // TODO: Move to Platform.Data
635            public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
636            {
637                var nullConstant = links.Constants.Null;
638                var equalityComparer = EqualityComparer<TLink>.Default;
639                var link = links.GetLink(linkIndex);
640                for (int i = 1; i < link.Count; i++)
641                {
642                    if (!equalityComparer.Equals(link[i], nullConstant))
643                    {
644                        return false;
645                    }
646                }
647                return true;
648            }
649
650            // TODO: Create a universal version of this method in Platform.Data (with using of for
                ↪  loop)
651            public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
652            {
653                var nullConstant = links.Constants.Null;
654                var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
655                links.Update(updateRequest);
656            }
657
658            // TODO: Create a universal version of this method in Platform.Data (with using of for
                ↪  loop)
659            public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
660            {
661                if (!links.AreValuesReset(linkIndex))
662                {
663                    links.ResetValues(linkIndex);
664                }
```

```
665            }
666
667            /// <summary>
668            /// Merging two usages graphs, all children of old link moved to be children of new link
669            ///   or deleted.
670            /// </summary>
671            public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
672                   TLink newLinkIndex)
673            {
674                var equalityComparer = EqualityComparer<TLink>.Default;
675                if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
676                {
677                    var constants = links.Constants;
678                    var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
679                           constants.Any);
680                    long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
681                    var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
682                           oldLinkIndex);
683                    long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
684                    var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
685                           usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
686                    if (!isStandalonePoint)
687                    {
688                        var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
689                        if (totalUsages > 0)
690                        {
691                            var usages = ArrayPool.Allocate<TLink>(totalUsages);
692                            var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
693                                   links.Constants.Continue);
694                            var i = 0L;
695                            if (usagesAsSourceCount > 0)
696                            {
697                                links.Each(usagesFiller.AddFirstAndReturnConstant,
698                                       usagesAsSourceQuery);
699                                for (; i < usagesAsSourceCount; i++)
700                                {
701                                    var usage = usages[i];
702                                    if (!equalityComparer.Equals(usage, oldLinkIndex))
703                                    {
704                                        links.Update(usage, newLinkIndex, links.GetTarget(usage));
705                                    }
706                                }
707                            }
708                            if (usagesAsTargetCount > 0)
709                            {
710                                links.Each(usagesFiller.AddFirstAndReturnConstant,
711                                       usagesAsTargetQuery);
712                                for (; i < usages.Length; i++)
713                                {
714                                    var usage = usages[i];
715                                    if (!equalityComparer.Equals(usage, oldLinkIndex))
716                                    {
717                                        links.Update(usage, links.GetSource(usage), newLinkIndex);
718                                    }
719                                }
720                            }
721                            ArrayPool.Free(usages);
722                        }
723                    }
724                }
725                return newLinkIndex;
726            }
727
728            /// <summary>
729            /// Replace one link with another (replaced link is deleted, children are updated or
730            ///   deleted).
731            /// </summary>
732            [MethodImpl(MethodImplOptions.AggressiveInlining)]
733            public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
734                   TLink newLinkIndex)
735            {
736                var equalityComparer = EqualityComparer<TLink>.Default;
737                if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
738                {
739                    links.MergeUsages(oldLinkIndex, newLinkIndex);
740                    links.Delete(oldLinkIndex);
741                }
742                return newLinkIndex;
```

```
733            }
734
735            public static ILinks<TLink>
               ↪  DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
736            {
737                links = new LinksCascadeUsagesResolver<TLink>(links);
738                links = new NonNullContentsLinkDeletionResolver<TLink>(links);
739                links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
740                return links;
741            }
742        }
743    }
```

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

```
1   using System.Collections.Generic;
2   using Platform.Interfaces;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Incrementers
7   {
8       public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
9       {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
               ↪  EqualityComparer<TLink>.Default;
11
12          private readonly TLink _frequencyMarker;
13          private readonly TLink _unaryOne;
14          private readonly IIncrementer<TLink> _unaryNumberIncrementer;
15
16          public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
               ↪  IIncrementer<TLink> unaryNumberIncrementer)
17              : base(links)
18          {
19              _frequencyMarker = frequencyMarker;
20              _unaryOne = unaryOne;
21              _unaryNumberIncrementer = unaryNumberIncrementer;
22          }
23
24          public TLink Increment(TLink frequency)
25          {
26              if (_equalityComparer.Equals(frequency, default))
27              {
28                  return Links.GetOrCreate(_unaryOne, _frequencyMarker);
29              }
30              var source = Links.GetSource(frequency);
31              var incrementedSource = _unaryNumberIncrementer.Increment(source);
32              return Links.GetOrCreate(incrementedSource, _frequencyMarker);
33          }
34      }
35  }
```

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

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

```csharp
                return Links.GetOrCreate(unaryNumber, _unaryOne);
            }
            else
            {
                return Links.GetOrCreate(source, Increment(target));
            }
        }
    }
}
```

./Platform.Data.Doublets/ISynchronizedLinks.cs

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

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

./Platform.Data.Doublets/Link.cs

```csharp
using Platform.Collections.Lists;
using Platform.Exceptions;
using Platform.Ranges;
using Platform.Singletons;
using System;
using System.Collections;
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
{
    /// <summary>
    /// Структура описывающая уникальную связь.
    /// </summary>
    public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
    {
        public static readonly Link<TLink> Null = new Link<TLink>();

        private static readonly LinksConstants<TLink> _constants =
        ↪  Default<LinksConstants<TLink>>.Instance;
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private const int Length = 3;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
        ↪  Target);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link(IList<TLink> values) => SetValues(values, out Index, out Source, out Target);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link(object other)
        {
            if (other is Link<TLink> otherLink)
            {
                SetValues(ref otherLink, out Index, out Source, out Target);
            }
            else if(other is IList<TLink> otherList)
            {
                SetValues(otherList, out Index, out Source, out Target);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
        ↪  Target);
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link(TLink index, TLink source, TLink target)
        {
            Index = index;
            Source = source;
            Target = target;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
        ↪    out TLink target)
        {
            index = other.Index;
            source = other.Source;
            target = other.Target;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void SetValues(IList<TLink> values, out TLink index, out TLink source,
        ↪    out TLink target)
        {
            switch (values.Count)
            {
                case 3:
                    index = values[0];
                    source = values[1];
                    target = values[2];
                    break;
                case 2:
                    index = values[0];
                    source = values[1];
                    target = default;
                    break;
                case 1:
                    index = values[0];
                    source = default;
                    target = default;
                    break;
                default:
                    index = default;
                    source = default;
                    target = default;
                    break;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (Index, Source, Target).GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
                            && _equalityComparer.Equals(Source, _constants.Null)
                            && _equalityComparer.Equals(Target, _constants.Null);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object other) => other is Link<TLink> &&
        ↪    Equals((Link<TLink>)other);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
                                        && _equalityComparer.Equals(Source, other.Source)
                                        && _equalityComparer.Equals(Target, other.Target);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
        ↪    {source}->{target})";

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string ToString(TLink source, TLink target) => $"({source}->{target})";

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

131        #region IList

133        public int Count => Length;

135        public bool IsReadOnly => true;

137        public TLink this[int index]
138        {
139            [MethodImpl(MethodImplOptions.AggressiveInlining)]
140            get
141            {
142                Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
                    ↪   nameof(index));
143                if (index == _constants.IndexPart)
144                {
145                    return Index;
146                }
147                if (index == _constants.SourcePart)
148                {
149                    return Source;
150                }
151                if (index == _constants.TargetPart)
152                {
153                    return Target;
154                }
155                throw new NotSupportedException(); // Impossible path due to
                    ↪   Ensure.ArgumentInRange
156            }
157            [MethodImpl(MethodImplOptions.AggressiveInlining)]
158            set => throw new NotSupportedException();
159        }

161        [MethodImpl(MethodImplOptions.AggressiveInlining)]
162        IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

164        [MethodImpl(MethodImplOptions.AggressiveInlining)]
165        public IEnumerator<TLink> GetEnumerator()
166        {
167            yield return Index;
168            yield return Source;
169            yield return Target;
170        }

172        [MethodImpl(MethodImplOptions.AggressiveInlining)]
173        public void Add(TLink item) => throw new NotSupportedException();

175        [MethodImpl(MethodImplOptions.AggressiveInlining)]
176        public void Clear() => throw new NotSupportedException();

178        [MethodImpl(MethodImplOptions.AggressiveInlining)]
179        public bool Contains(TLink item) => IndexOf(item) >= 0;

181        [MethodImpl(MethodImplOptions.AggressiveInlining)]
182        public void CopyTo(TLink[] array, int arrayIndex)
183        {
184            Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
185            Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                ↪   nameof(arrayIndex));
186            if (arrayIndex + Length > array.Length)
187            {
188                throw new InvalidOperationException();
189            }
190            array[arrayIndex++] = Index;
191            array[arrayIndex++] = Source;
192            array[arrayIndex] = Target;
193        }

195        [MethodImpl(MethodImplOptions.AggressiveInlining)]
196        public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();

198        [MethodImpl(MethodImplOptions.AggressiveInlining)]
199        public int IndexOf(TLink item)
200        {
201            if (_equalityComparer.Equals(Index, item))
202            {
203                return _constants.IndexPart;
```

```
204            }
205            if (_equalityComparer.Equals(Source, item))
206            {
207                return _constants.SourcePart;
208            }
209            if (_equalityComparer.Equals(Target, item))
210            {
211                return _constants.TargetPart;
212            }
213            return -1;
214        }
215
216        [MethodImpl(MethodImplOptions.AggressiveInlining)]
217        public void Insert(int index, TLink item) => throw new NotSupportedException();
218
219        [MethodImpl(MethodImplOptions.AggressiveInlining)]
220        public void RemoveAt(int index) => throw new NotSupportedException();
221
222        #endregion
223    }
224 }
```

./Platform.Data.Doublets/LinkExtensions.cs

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

./Platform.Data.Doublets/LinksOperatorBase.cs

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

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

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

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

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

```csharp
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Reflection;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
10     public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
       ↪  IConverter<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
13
14         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
15
16         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
           ↪  powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
           ↪  powerOf2ToUnaryNumberConverter;
17
18         public TLink Convert(TLink number)
19         {
20             var nullConstant = Links.Constants.Null;
21             var one = Integer<TLink>.One;
22             var target = nullConstant;
23             for (int i = 0; !_equalityComparer.Equals(number, default) && i <
               ↪  NumericType<TLink>.BitsLength; i++)
24             {
25                 if (_equalityComparer.Equals(Bit.And(number, one), one))
26                 {
27                     target = _equalityComparer.Equals(target, nullConstant)
28                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
29                         : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
30                 }
31                 number = Bit.ShiftRight(number, 1);
32             }
33             return target;
34         }
35     }
36  }
```

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

```
35            {
36                return default;
37            }
38            var frequencyNumber = Links.GetSource(frequency);
39            return _unaryNumberToAddressConverter.Convert(frequencyNumber);
40        }
41    }
42 }
```

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

```
1  using System.Collections.Generic;
2  using Platform.Exceptions;
3  using Platform.Interfaces;
4  using Platform.Ranges;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
10     public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<int, TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
13
14         private readonly TLink[] _unaryNumberPowersOf2;
15
16         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
17         {
18             _unaryNumberPowersOf2 = new TLink[64];
19             _unaryNumberPowersOf2[0] = one;
20         }
21
22         public TLink Convert(int power)
23         {
24             Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
               ↪  - 1), nameof(power));
25             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
26             {
27                 return _unaryNumberPowersOf2[power];
28             }
29             var previousPowerOf2 = Convert(power - 1);
30             var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
31             _unaryNumberPowersOf2[power] = powerOf2;
32             return powerOf2;
33         }
34     }
35 }
```

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

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

```
29                      { _unaryOne, one }
30                  };
31                  var unary = _unaryOne;
32                  var number = one;
33                  for (var i = 1; i < 64; i++)
34                  {
35                      unary = Links.GetOrCreate(unary, unary);
36                      number = Double(number);
37                      _unaryToUInt64.Add(unary, number);
38                  }
39              }
40
41          public TLink Convert(TLink unaryNumber)
42          {
43              if (_equalityComparer.Equals(unaryNumber, default))
44              {
45                  return default;
46              }
47              if (_equalityComparer.Equals(unaryNumber, _unaryOne))
48              {
49                  return Integer<TLink>.One;
50              }
51              var source = Links.GetSource(unaryNumber);
52              var target = Links.GetTarget(unaryNumber);
53              if (_equalityComparer.Equals(source, target))
54              {
55                  return _unaryToUInt64[unaryNumber];
56              }
57              else
58              {
59                  var result = _unaryToUInt64[source];
60                  TLink lastValue;
61                  while (!_unaryToUInt64.TryGetValue(target, out lastValue))
62                  {
63                      source = Links.GetSource(target);
64                      result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
65                      target = Links.GetTarget(target);
66                  }
67                  result = Arithmetic<TLink>.Add(result, lastValue);
68                  return result;
69              }
70          }
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          private static TLink Double(TLink number) => (Integer<TLink>)((Integer<TLink>)number *
    ↪  2UL);
74      }
75  }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Interfaces;
4   using Platform.Reflection;
5   using Platform.Numbers;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Numbers.Unary
10  {
11      public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<TLink>
12      {
13          private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪  EqualityComparer<TLink>.Default;
14
15          private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
16
17          public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
    ↪  TLink> powerOf2ToUnaryNumberConverter)
18              : base(links)
19          {
20              _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
21              for (int i = 0; i < NumericType<TLink>.BitsLength; i++)
22              {
23                  _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
24              }
25          }
26
```

```csharp
            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

```csharp
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

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

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

namespace Platform.Data.Doublets.PropertyOperators
```

```csharp
  7  {
  8      public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IPropertyOperator<TLink,
     ↪  TLink>
  9      {
 10          private static readonly EqualityComparer<TLink> _equalityComparer =
     ↪  EqualityComparer<TLink>.Default;
 11
 12          private readonly TLink _propertyMarker;
 13          private readonly TLink _propertyValueMarker;
 14
 15          public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
     ↪  propertyValueMarker) : base(links)
 16          {
 17              _propertyMarker = propertyMarker;
 18              _propertyValueMarker = propertyValueMarker;
 19          }
 20
 21          public TLink Get(TLink link)
 22          {
 23              var property = Links.SearchOrDefault(link, _propertyMarker);
 24              var container = GetContainer(property);
 25              var value = GetValue(container);
 26              return value;
 27          }
 28
 29          private TLink GetContainer(TLink property)
 30          {
 31              var valueContainer = default(TLink);
 32              if (_equalityComparer.Equals(property, default))
 33              {
 34                  return valueContainer;
 35              }
 36              var constants = Links.Constants;
 37              var countinueConstant = constants.Continue;
 38              var breakConstant = constants.Break;
 39              var anyConstant = constants.Any;
 40              var query = new Link<TLink>(anyConstant, property, anyConstant);
 41              Links.Each(candidate =>
 42              {
 43                  var candidateTarget = Links.GetTarget(candidate);
 44                  var valueTarget = Links.GetTarget(candidateTarget);
 45                  if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
 46                  {
 47                      valueContainer = Links.GetIndex(candidate);
 48                      return breakConstant;
 49                  }
 50                  return countinueConstant;
 51              }, query);
 52              return valueContainer;
 53          }
 54
 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/Generic/LinksAvlBalancedTreeMethodsBase.cs

```csharp
 1  using System;
 2  using System.Text;
 3  using System.Collections.Generic;
 4  using System.Runtime.CompilerServices;
 5  using Platform.Numbers;
 6  using Platform.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
```

```csharp
namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> :
        SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
    {
        protected readonly TLink Break;
        protected readonly TLink Continue;
        protected readonly byte* Links;
        protected readonly byte* Header;

        public LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
            byte* header)
        {
            Links = links;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
        {
            ref var link = ref GetLinkReference(linkIndex);
            return new Link<TLink>(linkIndex, link.Source, link.Target);
        }

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

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

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

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

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

```
 79             {
 80                 //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
 81                 return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
 82             }
 83         }
 84
 85         [MethodImpl(MethodImplOptions.AggressiveInlining)]
 86         protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
 87         {
 88             unchecked
 89             {
 90                 var previousValue = storedValue;
 91                 var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
                    ↪  1);
 92                 storedValue = modified;
 93             }
 94         }
 95
 96         [MethodImpl(MethodImplOptions.AggressiveInlining)]
 97         protected virtual bool GetRightIsChildValue(TLink value)
 98         {
 99             unchecked
100             {
101                 //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
102                 return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
103             }
104         }
105
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
108         {
109             unchecked
110             {
111                 var previousValue = storedValue;
112                 var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
                    ↪  1);
113                 storedValue = modified;
114             }
115         }
116
117         [MethodImpl(MethodImplOptions.AggressiveInlining)]
118         protected virtual sbyte GetBalanceValue(TLink storedValue)
119         {
120             unchecked
121             {
122                 var value = (int)(Integer<TLink>)Bit<TLink>.PartialRead(storedValue, 0, 3);
123                 value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
                    ↪  end of sbyte
124                 return (sbyte)value;
125             }
126         }
127
128         [MethodImpl(MethodImplOptions.AggressiveInlining)]
129         protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
130         {
131             unchecked
132             {
133                 var packagedValue = (TLink)(Integer<TLink>)((byte)value >> 5 & 4 | value & 3);
134                 var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
135                 storedValue = modified;
136             }
137         }
138
139         public TLink this[TLink index]
140         {
141             get
142             {
143                 var root = GetTreeRoot();
144                 if (GreaterOrEqualThan(index, GetSize(root)))
145                 {
146                     return Zero;
147                 }
148                 while (!EqualToZero(root))
149                 {
150                     var left = GetLeftOrDefault(root);
151                     var leftSize = GetSizeOrZero(left);
152                     if (LessThan(index, leftSize))
153                     {
154                         root = left;
```

```
155                        continue;
156                    }
157                    if (IsEquals(index, leftSize))
158                    {
159                        return root;
160                    }
161                    root = GetRightOrDefault(root);
162                    index = Subtract(index, Increment(leftSize));
163                }
164                return Zero; // TODO: Impossible situation exception (only if tree structure
    ↪    broken)
165            }
166        }
167
168        /// <summary>
169        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
    ↪    (концом).
170        /// </summary>
171        /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
172        /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
173        /// <returns>Индекс искомой связи.</returns>
174        public TLink Search(TLink source, TLink target)
175        {
176            var root = GetTreeRoot();
177            while (!EqualToZero(root))
178            {
179                ref var rootLink = ref GetLinkReference(root);
180                var rootSource = rootLink.Source;
181                var rootTarget = rootLink.Target;
182                if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
    ↪    node.Key < root.Key
183                {
184                    root = GetLeftOrDefault(root);
185                }
186                else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
    ↪    node.Key > root.Key
187                {
188                    root = GetRightOrDefault(root);
189                }
190                else // node.Key == root.Key
191                {
192                    return root;
193                }
194            }
195            return Zero;
196        }
197
198        // TODO: Return indices range instead of references count
199        public TLink CountUsages(TLink link)
200        {
201            var root = GetTreeRoot();
202            var total = GetSize(root);
203            var totalRightIgnore = Zero;
204            while (!EqualToZero(root))
205            {
206                var @base = GetBasePartValue(root);
207                if (LessOrEqualThan(@base, link))
208                {
209                    root = GetRightOrDefault(root);
210                }
211                else
212                {
213                    totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
214                    root = GetLeftOrDefault(root);
215                }
216            }
217            root = GetTreeRoot();
218            var totalLeftIgnore = Zero;
219            while (!EqualToZero(root))
220            {
221                var @base = GetBasePartValue(root);
222                if (GreaterOrEqualThan(@base, link))
223                {
224                    root = GetLeftOrDefault(root);
225                }
226                else
227                {
228                    totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
```

```
229
230                      root = GetRightOrDefault(root);
231                  }
232              }
233              return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
234          }
235
236          public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
237          {
238              var root = GetTreeRoot();
239              if (EqualToZero(root))
240              {
241                  return Continue;
242              }
243              TLink first = Zero, current = root;
244              while (!EqualToZero(current))
245              {
246                  var @base = GetBasePartValue(current);
247                  if (GreaterOrEqualThan(@base, link))
248                  {
249                      if (IsEquals(@base, link))
250                      {
251                          first = current;
252                      }
253                      current = GetLeftOrDefault(current);
254                  }
255                  else
256                  {
257                      current = GetRightOrDefault(current);
258                  }
259              }
260              if (!EqualToZero(first))
261              {
262                  current = first;
263                  while (true)
264                  {
265                      if (IsEquals(handler(GetLinkValues(current)), Break))
266                      {
267                          return Break;
268                      }
269                      current = GetNext(current);
270                      if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
271                      {
272                          break;
273                      }
274                  }
275              }
276              return Continue;
277          }
278
279          protected override void PrintNodeValue(TLink node, StringBuilder sb)
280          {
281              ref var link = ref GetLinkReference(node);
282              sb.Append(' ');
283              sb.Append(link.Source);
284              sb.Append('-');
285              sb.Append('>');
286              sb.Append(link.Target);
287          }
288      }
289  }
```

./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSizeBalancedTreeMethodsBase.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.Generic
12  {
13      public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> :
      ↪  SizeBalancedTreeMethods2<TLink>, ILinksTreeMethods<TLink>
14      {
15          protected readonly TLink Break;
16          protected readonly TLink Continue;
```

```csharp
        protected readonly byte* Links;
        protected readonly byte* Header;

        public LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
            byte* header)
        {
            Links = links;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
        {
            ref var link = ref GetLinkReference(linkIndex);
            return new Link<TLink>(linkIndex, link.Source, link.Target);
        }

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

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

        public TLink this[TLink index]
        {
            get
            {
                var root = GetTreeRoot();
                if (GreaterOrEqualThan(index, GetSize(root)))
                {
                    return Zero;
                }
                while (!EqualToZero(root))
                {
                    var left = GetLeftOrDefault(root);
                    var leftSize = GetSizeOrZero(left);
                    if (LessThan(index, leftSize))
                    {
                        root = left;
                        continue;
                    }
                    if (IsEquals(index, leftSize))
                    {
```

```csharp
                            return root;
                        }
                        root = GetRightOrDefault(root);
                        index = Subtract(index, Increment(leftSize));
                    }
                    return Zero; // TODO: Impossible situation exception (only if tree structure
                    ↪   broken)
                }
            }

            /// <summary>
            /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
            ↪   (концом).
            /// </summary>
            /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
            /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
            /// <returns>Индекс искомой связи.</returns>
            public TLink Search(TLink source, TLink target)
            {
                var root = GetTreeRoot();
                while (!EqualToZero(root))
                {
                    ref var rootLink = ref GetLinkReference(root);
                    var rootSource = rootLink.Source;
                    var rootTarget = rootLink.Target;
                    if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                    ↪   node.Key < root.Key
                    {
                        root = GetLeftOrDefault(root);
                    }
                    else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                    ↪   node.Key > root.Key
                    {
                        root = GetRightOrDefault(root);
                    }
                    else // node.Key == root.Key
                    {
                        return root;
                    }
                }
                return Zero;
            }

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

                        root = GetRightOrDefault(root);
                    }
                }
```

```csharp
                    return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
            ↪  EachUsageCore(@base, GetTreeRoot(), handler);

            // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
            ↪  low-level MSIL stack.
            private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
            {
                var @continue = Continue;
                if (EqualToZero(link))
                {
                    return @continue;
                }
                var linkBasePart = GetBasePartValue(link);
                var @break = Break;
                if (GreaterThan(linkBasePart, @base))
                {
                    if (IsEquals(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
                    {
                        return @break;
                    }
                }
                else if (LessThan(linkBasePart, @base))
                {
                    if (IsEquals(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
                    {
                        return @break;
                    }
                }
                else //if (linkBasePart == @base)
                {
                    if (IsEquals(handler(GetLinkValues(link)), @break))
                    {
                        return @break;
                    }
                    if (IsEquals(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
                    {
                        return @break;
                    }
                    if (IsEquals(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
                    {
                        return @break;
                    }
                }
                return @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/Generic/LinksSourcesAvlBalancedTreeMethods.cs

```csharp
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe class LinksSourcesAvlBalancedTreeMethods<TLink> :
    ↪  LinksAvlBalancedTreeMethodsBase<TLink>
    {
        public LinksSourcesAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
        ↪  byte* header) : base(constants, links, header) { }

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

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected unsafe override ref TLink GetRightReference(TLink node) => ref
        ↪  GetLinkReference(node).RightAsSource;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe class LinksSourcesSizeBalancedTreeMethods<TLink> :
        LinksSizeBalancedTreeMethodsBase<TLink>
    {
        public LinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
            byte* header) : base(constants, links, header) { }

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe class LinksTargetsAvlBalancedTreeMethods<TLink> :
        LinksAvlBalancedTreeMethodsBase<TLink>
    {
```

```csharp
        public LinksTargetsAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
        ↪  byte* header) : base(constants, 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)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
        ↪  IsEquals(firstTarget, secondTarget) && LessThan(firstSource, secondSource);

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

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

```
70              link.RightAsTarget = Zero;
71              link.SizeAsTarget = Zero;
72          }
73      }
74  }
```

./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsSizeBalancedTreeMethods.cs

```csharp
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.Generic
6   {
7       public unsafe class LinksTargetsSizeBalancedTreeMethods<TLink> :
        ↪  LinksSizeBalancedTreeMethodsBase<TLink>
8       {
9           public LinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
            ↪  byte* header) : base(constants, links, header) { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected unsafe override ref TLink GetLeftReference(TLink node) => ref
            ↪  GetLinkReference(node).LeftAsTarget;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected unsafe override ref TLink GetRightReference(TLink node) => ref
            ↪  GetLinkReference(node).RightAsTarget;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override void SetLeft(TLink node, TLink left) =>
            ↪  GetLinkReference(node).LeftAsTarget = left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(TLink node, TLink right) =>
            ↪  GetLinkReference(node).RightAsTarget = right;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override void SetSize(TLink node, TLink size) =>
            ↪  GetLinkReference(node).SizeAsTarget = size;
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
            ↪  IsEquals(firstTarget, secondTarget) && LessThan(firstSource, secondSource);
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
            ↪  IsEquals(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          protected override void ClearNode(TLink node)
49          {
50              ref var link = ref GetLinkReference(node);
51              link.LeftAsTarget = Zero;
52              link.RightAsTarget = Zero;
53              link.SizeAsTarget = Zero;
54          }
55      }
56  }
```

./Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinksBase.cs

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

```csharp
using Platform.Collections.Arrays;
using Platform.Data.Exceptions;
using Platform.Disposables;
using Platform.Memory;
using Platform.Numbers;
using Platform.Singletons;

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

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public abstract class ResizableDirectMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
    {
        protected static readonly EqualityComparer<TLink> EqualityComparer =
            EqualityComparer<TLink>.Default;
        protected static readonly Comparer<TLink> Comparer = Comparer<TLink>.Default;

        /// <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 long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;

        public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;

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

        protected readonly IResizableDirectMemory _memory;
        protected readonly long _memoryReservationStep;

        protected ILinksTreeMethods<TLink> TargetsTreeMethods;
        protected ILinksTreeMethods<TLink> SourcesTreeMethods;
        // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
        //   нужно использовать не список а дерево, так как так можно быстрее проверить на
        //   наличие связи внутри
        protected ILinksListMethods<TLink> UnusedLinksListMethods;

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

        public virtual LinksConstants<TLink> Constants { get; }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
            memoryReservationStep, LinksConstants<TLink> constants)
        {
            _memory = memory;
            _memoryReservationStep = memoryReservationStep;
            Constants = constants;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
            memoryReservationStep) : this(memory, memoryReservationStep,
            Default<LinksConstants<TLink>>.Instance) { }

        protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
        {
            if (memory.ReservedCapacity < memoryReservationStep)
            {
                memory.ReservedCapacity = memoryReservationStep;
            }
            SetPointers(_memory);
            ref var header = ref GetHeaderReference();
            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
            _memory.UsedCapacity = ConvertToUInt64(header.AllocatedLinks) * LinkSizeInBytes +
                LinkHeaderSizeInBytes;
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            header.ReservedLinks = ConvertToAddress((_memory.ReservedCapacity -
                LinkHeaderSizeInBytes) / LinkSizeInBytes);
        }
```

```csharp
76              }

78              [MethodImpl(MethodImplOptions.AggressiveInlining)]
79              public virtual TLink Count(IList<TLink> restrictions)
80              {
81                  // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
82                  if (restrictions.Count == 0)
83                  {
84                      return Total;
85                  }
86                  var constants = Constants;
87                  var any = constants.Any;
88                  var index = restrictions[constants.IndexPart];
89                  if (restrictions.Count == 1)
90                  {
91                      if (AreEqual(index, any))
92                      {
93                          return Total;
94                      }
95                      return Exists(index) ? GetOne() : GetZero();
96                  }
97                  if (restrictions.Count == 2)
98                  {
99                      var value = restrictions[1];
100                     if (AreEqual(index, any))
101                     {
102                         if (AreEqual(value, any))
103                         {
104                             return Total; // Any - как отсутствие ограничения
105                         }
106                         return Add(SourcesTreeMethods.CountUsages(value),
                        ↪   TargetsTreeMethods.CountUsages(value));
107                     }
108                     else
109                     {
110                         if (!Exists(index))
111                         {
112                             return GetZero();
113                         }
114                         if (AreEqual(value, any))
115                         {
116                             return GetOne();
117                         }
118                         ref var storedLinkValue = ref GetLinkReference(index);
119                         if (AreEqual(storedLinkValue.Source, value) ||
                        ↪   AreEqual(storedLinkValue.Target, value))
120                         {
121                             return GetOne();
122                         }
123                         return GetZero();
124                     }
125                 }
126                 if (restrictions.Count == 3)
127                 {
128                     var source = restrictions[constants.SourcePart];
129                     var target = restrictions[constants.TargetPart];
130                     if (AreEqual(index, any))
131                     {
132                         if (AreEqual(source, any) && AreEqual(target, any))
133                         {
134                             return Total;
135                         }
136                         else if (AreEqual(source, any))
137                         {
138                             return TargetsTreeMethods.CountUsages(target);
139                         }
140                         else if (AreEqual(target, any))
141                         {
142                             return SourcesTreeMethods.CountUsages(source);
143                         }
144                         else //if(source != Any && target != Any)
145                         {
146                             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
147                             var link = SourcesTreeMethods.Search(source, target);
148                             return AreEqual(link, constants.Null) ? GetZero() : GetOne();
149                         }
150                     }
151                     else
```

```csharp
                {
                    if (!Exists(index))
                    {
                        return GetZero();
                    }
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return GetOne();
                    }
                    ref var storedLinkValue = ref GetLinkReference(index);
                    if (!AreEqual(source, any) && !AreEqual(target, any))
                    {
                        if (AreEqual(storedLinkValue.Source, source) &&
                        ↪  AreEqual(storedLinkValue.Target, target))
                        {
                            return GetOne();
                        }
                        return GetZero();
                    }
                    var value = default(TLink);
                    if (AreEqual(source, any))
                    {
                        value = target;
                    }
                    if (AreEqual(target, any))
                    {
                        value = source;
                    }
                    if (AreEqual(storedLinkValue.Source, value) ||
                    ↪  AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪  поддерживаются.");
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            var constants = Constants;
            var @break = constants.Break;
            if (restrictions.Count == 0)
            {
                for (var link = GetOne(); LessOrEqualThan(link,
                ↪  GetHeaderReference().AllocatedLinks); link = Increment(link))
                {
                    if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
                    {
                        return @break;
                    }
                }
                return @break;
            }
            var @continue = constants.Continue;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Each(handler, GetEmptyList());
                }
                if (!Exists(index))
                {
                    return @continue;
                }
                return handler(GetLinkStruct(index));
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
```

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

```
303            if (AreEqual(target, any))
304            {
305                value = source;
306            }
307            if (AreEqual(storedLinkValue.Source, value) ||
308                AreEqual(storedLinkValue.Target, value))
309            {
310                return handler(GetLinkStruct(index));
311            }
312            return @continue;
313        }
314    }
315    throw new NotSupportedException("Другие размеры и способы ограничений не
    ↪  поддерживаются.");
316 }

317
318 /// <remarks>
319 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↪  в другом месте (но не в менеджере памяти, а в логике Links)
320 /// </remarks>
321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
322 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
323 {
324    var constants = Constants;
325    var @null = constants.Null;
326    var linkIndex = restrictions[constants.IndexPart];
327    ref var link = ref GetLinkReference(linkIndex);
328    ref var header = ref GetHeaderReference();
329    ref var firstAsSource = ref header.FirstAsSource;
330    ref var firstAsTarget = ref header.FirstAsTarget;
331    // Будет корректно работать только в том случае, если пространство выделенной связи
    ↪  предварительно заполнено нулями
332    if (!AreEqual(link.Source, @null))
333    {
334        SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
335    }
336    if (!AreEqual(link.Target, @null))
337    {
338        TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
339    }
340    link.Source = substitution[constants.SourcePart];
341    link.Target = substitution[constants.TargetPart];
342    if (!AreEqual(link.Source, @null))
343    {
344        SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
345    }
346    if (!AreEqual(link.Target, @null))
347    {
348        TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
349    }
350    return linkIndex;
351 }

352
353 /// <remarks>
354 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
    ↪  пространство
355 /// </remarks>
356 public virtual TLink Create(IList<TLink> restrictions)
357 {
358    ref var header = ref GetHeaderReference();
359    var freeLink = header.FirstFreeLink;
360    if (!AreEqual(freeLink, Constants.Null))
361    {
362        UnusedLinksListMethods.Detach(freeLink);
363    }
364    else
365    {
366        var maximumPossibleInnerReference =
        ↪  Constants.PossibleInnerReferencesRange.Maximum;
367        if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
368        {
369            throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
370        }
371        if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
372        {
373            _memory.ReservedCapacity += _memoryReservationStep;
374            SetPointers(_memory);
375            header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
            ↪  LinkSizeInBytes);
```

```
376                    }
377                    header.AllocatedLinks = Increment(header.AllocatedLinks);
378                    _memory.UsedCapacity += LinkSizeInBytes;
379                    freeLink = header.AllocatedLinks;
380                }
381                return freeLink;
382            }
383
384            [MethodImpl(MethodImplOptions.AggressiveInlining)]
385            public virtual void Delete(IList<TLink> restrictions)
386            {
387                ref var header = ref GetHeaderReference();
388                var link = restrictions[Constants.IndexPart];
389                if (LessThan(link, header.AllocatedLinks))
390                {
391                    UnusedLinksListMethods.AttachAsFirst(link);
392                }
393                else if (AreEqual(link, header.AllocatedLinks))
394                {
395                    header.AllocatedLinks = Decrement(header.AllocatedLinks);
396                    _memory.UsedCapacity -= LinkSizeInBytes;
397                    // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪  пока не дойдём до первой существующей связи
398                    // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
399                    while (GreaterThan(header.AllocatedLinks, GetZero()) &&
                    ↪  IsUnusedLink(header.AllocatedLinks))
400                    {
401                        UnusedLinksListMethods.Detach(header.AllocatedLinks);
402                        header.AllocatedLinks = Decrement(header.AllocatedLinks);
403                        _memory.UsedCapacity -= LinkSizeInBytes;
404                    }
405                }
406            }
407
408            [MethodImpl(MethodImplOptions.AggressiveInlining)]
409            public IList<TLink> GetLinkStruct(TLink linkIndex)
410            {
411                ref var link = ref GetLinkReference(linkIndex);
412                return new Link<TLink>(linkIndex, link.Source, link.Target);
413            }
414
415            /// <remarks>
416            /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
                ↪  адрес реально поменялся
417            ///
418            /// Указатель this.links может быть в том же месте,
419            /// так как 0-я связь не используется и имеет такой же размер как Header,
420            /// поэтому header размещается в том же месте, что и 0-я связь
421            /// </remarks>
422            [MethodImpl(MethodImplOptions.AggressiveInlining)]
423            protected abstract void SetPointers(IResizableDirectMemory memory);
424
425            [MethodImpl(MethodImplOptions.AggressiveInlining)]
426            protected virtual void ResetPointers()
427            {
428                SourcesTreeMethods = null;
429                TargetsTreeMethods = null;
430                UnusedLinksListMethods = null;
431            }
432
433            [MethodImpl(MethodImplOptions.AggressiveInlining)]
434            protected abstract ref LinksHeader<TLink> GetHeaderReference();
435
436            [MethodImpl(MethodImplOptions.AggressiveInlining)]
437            protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
438
439            [MethodImpl(MethodImplOptions.AggressiveInlining)]
440            protected virtual bool Exists(TLink link)
441                => GreaterOrEqualThan(link, Constants.PossibleInnerReferencesRange.Minimum)
442                && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
443                && !IsUnusedLink(link);
444
445            [MethodImpl(MethodImplOptions.AggressiveInlining)]
446            protected virtual bool IsUnusedLink(TLink linkIndex)
447            {
448                if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
                    ↪  is not needed
449                {
450                    ref var link = ref GetLinkReference(linkIndex);
```

```csharp
                    return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
                }
                else
                {
                    return true;
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink GetOne() => Integer<TLink>.One;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink GetZero() => Integer<TLink>.Zero;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool AreEqual(TLink first, TLink second) =>
          EqualityComparer.Equals(first, second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessThan(TLink first, TLink second) => Comparer.Compare(first,
          second) < 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
          Comparer.Compare(first, second) <= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterThan(TLink first, TLink second) => Comparer.Compare(first,
          second) > 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
          Comparer.Compare(first, second) >= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual long ConvertToUInt64(TLink value) => (Integer<TLink>)value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink ConvertToAddress(long value) => (Integer<TLink>)value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
          second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink Subtract(TLink first, TLink second) =>
          Arithmetic<TLink>.Subtract(first, second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLink> GetEmptyList() => ArrayPool<TLink>.Empty;

        #region Disposable

        protected override bool AllowMultipleDisposeCalls => true;

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

        #endregion
    }
}
```

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

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

```csharp
using System;
using Platform.Singletons;

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

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe partial class ResizableDirectMemoryLinks<TLink> :
        ResizableDirectMemoryLinksBase<TLink>
    {
        private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
        private byte* _header;
        private byte* _links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ResizableDirectMemoryLinks(string address) : this(address, DefaultLinksSizeStep)
            { }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
            DefaultLinksSizeStep) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            memoryReservationStep) : this(memory, memoryReservationStep,
            Default<LinksConstants<TLink>>.Instance, true) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            memoryReservationStep, LinksConstants<TLink> constants, bool useAvlBasedIndex) :
            base(memory, memoryReservationStep, constants)
        {
            if (useAvlBasedIndex)
            {
                _createSourceTreeMethods = () => new
                    LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
                _createTargetTreeMethods = () => new
                    LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
            }
            else
            {
                _createSourceTreeMethods = () => new
                    LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
                _createTargetTreeMethods = () => new
                    LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
            }
            Init(memory, memoryReservationStep);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPointers(IResizableDirectMemory memory)
        {
            _links = (byte*)memory.Pointer;
            _header = _links;
            SourcesTreeMethods = _createSourceTreeMethods();
            TargetsTreeMethods = _createTargetTreeMethods();
            UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ResetPointers()
        {
            base.ResetPointers();
            _links = null;
            _header = null;
```

```
68            }
69
70            [MethodImpl(MethodImplOptions.AggressiveInlining)]
71            protected override ref LinksHeader<TLink> GetHeaderReference() => ref
   ↪   AsRef<LinksHeader<TLink>>(_header);
72
73            [MethodImpl(MethodImplOptions.AggressiveInlining)]
74            protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
   ↪   AsRef<RawLink<TLink>>(_links + LinkSizeInBytes * (Integer<TLink>)linkIndex);
75        }
76    }
```

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

```
1    using System.Runtime.CompilerServices;
2    using Platform.Collections.Methods.Lists;
3    using Platform.Numbers;
4    using static System.Runtime.CompilerServices.Unsafe;
5
6    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8    namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
9    {
10        public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
    ↪   ILinksListMethods<TLink>
11        {
12            private readonly byte* _links;
13            private readonly byte* _header;
14
15            [MethodImpl(MethodImplOptions.AggressiveInlining)]
16            public UnusedLinksListMethods(byte* links, byte* header)
17            {
18                _links = links;
19                _header = header;
20            }
21
22            [MethodImpl(MethodImplOptions.AggressiveInlining)]
23            protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
    ↪   AsRef<LinksHeader<TLink>>(_header);
24
25            [MethodImpl(MethodImplOptions.AggressiveInlining)]
26            protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
    ↪   AsRef<RawLink<TLink>>(_links + RawLink<TLink>.SizeInBytes * (Integer<TLink>)link);
27
28            [MethodImpl(MethodImplOptions.AggressiveInlining)]
29            protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
30
31            [MethodImpl(MethodImplOptions.AggressiveInlining)]
32            protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
33
34            [MethodImpl(MethodImplOptions.AggressiveInlining)]
35            protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;
36
37            [MethodImpl(MethodImplOptions.AggressiveInlining)]
38            protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;
39
40            [MethodImpl(MethodImplOptions.AggressiveInlining)]
41            protected override TLink GetSize() => GetHeaderReference().FreeLinks;
42
43            [MethodImpl(MethodImplOptions.AggressiveInlining)]
44            protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
    ↪   element;
45
46            [MethodImpl(MethodImplOptions.AggressiveInlining)]
47            protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
    ↪   element;
48
49            [MethodImpl(MethodImplOptions.AggressiveInlining)]
50            protected override void SetPrevious(TLink element, TLink previous) =>
    ↪   GetLinkReference(element).Source = previous;
51
52            [MethodImpl(MethodImplOptions.AggressiveInlining)]
53            protected override void SetNext(TLink element, TLink next) =>
    ↪   GetLinkReference(element).Target = next;
54
55            [MethodImpl(MethodImplOptions.AggressiveInlining)]
56            protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
57        }
58    }
```

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

```csharp
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

```csharp
1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.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/LinksHeader.cs

```csharp
1  using Platform.Unsafe;
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 struct LinksHeader<TLink>
8      {
9          public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;
10
11         public TLink AllocatedLinks;
12         public TLink ReservedLinks;
13         public TLink FreeLinks;
14         public TLink FirstFreeLink;
15         public TLink FirstAsSource;
16         public TLink FirstAsTarget;
17         public TLink LastFreeLink;
18         public TLink Reserved8;
19     }
20 }
```

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

```csharp
1  using Platform.Unsafe;
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 struct RawLink<TLink>
8      {
9          public static readonly long SizeInBytes = Structure<RawLink<TLink>>.Size;
10
11         public TLink Source;
12         public TLink Target;
13         public TLink LeftAsSource;
14         public TLink RightAsSource;
15         public TLink SizeAsSource;
16         public TLink LeftAsTarget;
17         public TLink RightAsTarget;
18         public TLink SizeAsTarget;
19     }
20 }
```

./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs

```csharp
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.ResizableDirectMemory.Generic;
3  using static System.Runtime.CompilerServices.Unsafe;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
```

```csharp
namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
{
    public unsafe abstract class UInt64LinksAvlBalancedTreeMethodsBase :
        LinksAvlBalancedTreeMethodsBase<ulong>
    {
        protected new readonly RawLink<ulong>* Links;
        protected new readonly LinksHeader<ulong>* Header;

        public UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
            RawLink<ulong>* links, LinksHeader<ulong>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

        [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;

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
```

```
79              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
      ↪   secondLink.Source, secondLink.Target);
80          }
81
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          protected override ulong GetSizeValue(ulong value) => unchecked((value & 4294967264UL)
      ↪   >> 5);
84
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
      ↪   unchecked(storedValue & 31UL | (size & 134217727UL) << 5);
87
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          protected override bool GetLeftIsChildValue(ulong value) => unchecked((value & 16UL) >>
      ↪   4 == 1UL);
90
91          [MethodImpl(MethodImplOptions.AggressiveInlining)]
92          protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
      ↪   storedValue = unchecked(storedValue & 4294967279UL | (As<bool, byte>(ref value) &
      ↪   1UL) << 4);
93
94          [MethodImpl(MethodImplOptions.AggressiveInlining)]
95          protected override bool GetRightIsChildValue(ulong value) => unchecked((value & 8UL) >>
      ↪   3 == 1UL);
96
97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
98          protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
      ↪   storedValue = unchecked(storedValue & 4294967287UL | (As<bool, byte>(ref value) &
      ↪   1UL) << 3);
99
100         [MethodImpl(MethodImplOptions.AggressiveInlining)]
101         protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
      ↪   0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
      ↪   sbyte
102
103         [MethodImpl(MethodImplOptions.AggressiveInlining)]
104         protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
      ↪   storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
      ↪   value & 3) & 7UL);
105
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
108
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
111     }
112 }
```

./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs

```
1   using System.Runtime.CompilerServices;
2   using Platform.Data.Doublets.ResizableDirectMemory.Generic;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
7   {
8       public unsafe abstract class UInt64LinksSizeBalancedTreeMethodsBase :
      ↪   LinksSizeBalancedTreeMethodsBase<ulong>
9       {
10          protected new readonly RawLink<ulong>* Links;
11          protected new readonly LinksHeader<ulong>* Header;
12
13          public UInt64LinksSizeBalancedTreeMethodsBase(LinksConstants<ulong> constants,
      ↪   RawLink<ulong>* links, LinksHeader<ulong>* header)
14              : base(constants, (byte*)links, (byte*)header)
15          {
16              Links = links;
17              Header = header;
18          }
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override ulong GetZero() => 0UL;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override bool EqualToZero(ulong value) => value == 0UL;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override bool IsEquals(ulong first, ulong second) => first == second;
28
```

```csharp
        [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;

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
    }
}
```

./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs

```csharp
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
{
    public unsafe class UInt64LinksSourcesAvlBalancedTreeMethods :
    ↪  UInt64LinksAvlBalancedTreeMethodsBase
    {
        public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↪  { }
```

```csharp
        [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);

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

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

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

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

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

```
 74              }
 75          }
 76    }
```

./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.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.Specific
  6    {
  7        public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
           ↪  UInt64LinksSizeBalancedTreeMethodsBase
  8        {
  9            public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
               ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
               ↪  { }
 10
 11            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 12            protected override ref ulong GetLeftReference(ulong node) => ref
               ↪  Links[node].LeftAsSource;
 13
 14            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 15            protected override ref ulong GetRightReference(ulong node) => ref
               ↪  Links[node].RightAsSource;
 16
 17            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 18            protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
 19
 20            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 21            protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
 22
 23            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 24            protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
               ↪  left;
 25
 26            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 27            protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
               ↪  right;
 28
 29            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 30            protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;
 31
 32            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 33            protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
               ↪  size;
 34
 35            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 36            protected override ulong GetTreeRoot() => Header->FirstAsSource;
 37
 38            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 39            protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
 40
 41            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 42            protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
               ↪  ulong secondSource, ulong secondTarget)
 43                => firstSource < secondSource || firstSource == secondSource && firstTarget <
                   ↪  secondTarget;
 44
 45            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 46            protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
               ↪  ulong secondSource, ulong secondTarget)
 47                => firstSource > secondSource || firstSource == secondSource && firstTarget >
                   ↪  secondTarget;
 48
 49            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 50            protected override void ClearNode(ulong node)
 51            {
 52                ref var link = ref Links[node];
 53                link.LeftAsSource = 0UL;
 54                link.RightAsSource = 0UL;
 55                link.SizeAsSource = 0UL;
 56            }
 57        }
 58    }
```

./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs

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

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

namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
{
    public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
        UInt64LinksAvlBalancedTreeMethodsBase
    {
        public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
            RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
            { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
            Links[node].LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
            Links[node].RightAsTarget;

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

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

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

        [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)]
```

```
64        protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
     ↪ ulong secondSource, ulong secondTarget)
65            => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
     ↪ secondSource;
66
67        [MethodImpl(MethodImplOptions.AggressiveInlining)]
68        protected override void ClearNode(ulong node)
69        {
70            ref var link = ref Links[node];
71            link.LeftAsTarget = 0UL;
72            link.RightAsTarget = 0UL;
73            link.SizeAsTarget = 0UL;
74        }
75    }
76 }
```

./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.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.Specific
6  {
7      public unsafe class UInt64LinksTargetsSizeBalancedTreeMethods :
     ↪ UInt64LinksSizeBalancedTreeMethodsBase
8      {
9          public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
     ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
     ↪ { }
10
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected override ref ulong GetLeftReference(ulong node) => ref
     ↪ Links[node].LeftAsTarget;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetRightReference(ulong node) => ref
     ↪ Links[node].RightAsTarget;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
     ↪ left;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
     ↪ right;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
     ↪ size;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override ulong GetTreeRoot() => Header->FirstAsTarget;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
     ↪ ulong secondSource, ulong secondTarget)
43             => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
     ↪ secondSource;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
     ↪ ulong secondSource, ulong secondTarget)
47             => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
     ↪ secondSource;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override void ClearNode(ulong node)
```

```
51          {
52              ref var link = ref Links[node];
53              link.LeftAsTarget = 0UL;
54              link.RightAsTarget = 0UL;
55              link.SizeAsTarget = 0UL;
56          }
57      }
58  }
```

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

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Runtime.CompilerServices;
 4  using Platform.Memory;
 5  using Platform.Data.Doublets.ResizableDirectMemory.Generic;
 6  using Platform.Singletons;
 7
 8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 9
10  namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
11  {
12      public unsafe class UInt64ResizableDirectMemoryLinks : ResizableDirectMemoryLinksBase<ulong>
13      {
14          private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
15          private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
16          private LinksHeader<ulong>* _header;
17          private RawLink<ulong>* _links;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public UInt64ResizableDirectMemoryLinks(string address) : this(address,
            ↪  DefaultLinksSizeStep) { }
21
22          /// <summary>
23          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪   минимальным шагом расширения базы данных.
24          /// </summary>
25          /// <param name="address">Полный пусть к файлу базы данных.</param>
26          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪   байтах.</param>
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
            ↪   this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪   memoryReservationStep) { }
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪   DefaultLinksSizeStep) { }
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪   memoryReservationStep) : this(memory, memoryReservationStep,
            ↪   Default<LinksConstants<ulong>>.Instance, true) { }
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪   memoryReservationStep, LinksConstants<ulong> constants, bool useAvlBasedIndex) :
            ↪   base(memory, memoryReservationStep, constants)
38          {
39              if (useAvlBasedIndex)
40              {
41                  _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
42                  _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
43              }
44              else
45              {
46                  _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
47                  _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
48              }
49              Init(memory, memoryReservationStep);
50          }
51
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
53          protected override void SetPointers(IResizableDirectMemory memory)
54          {
55              _header = (LinksHeader<ulong>*)memory.Pointer;
```

```
            _links = (RawLink<ulong>*)memory.Pointer;
            SourcesTreeMethods = _createSourceTreeMethods();
            TargetsTreeMethods = _createTargetTreeMethods();
            UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ResetPointers()
        {
            base.ResetPointers();
            _links = null;
            _header = null;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
        ↪   _links[linkIndex];

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

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

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

        [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 ulong GetZero() => 0UL;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override long ConvertToUInt64(ulong value) => (long)value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong ConvertToAddress(long value) => (ulong)value;

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IList<ulong> GetEmptyList() => new ulong[0];
    }
}
```

./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64UnusedLinksListMethods.cs
```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.ResizableDirectMemory.Generic;

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

namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
{
    public unsafe class UInt64UnusedLinksListMethods : UnusedLinksListMethods<ulong>
    {
        private readonly RawLink<ulong>* _links;
        private readonly LinksHeader<ulong>* _header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnusedLinksListMethods(RawLink<ulong>* links, LinksHeader<ulong>* header)
```

```
15                : base((byte*)links, (byte*)header)
16            {
17                _links = links;
18                _header = header;
19            }
20
21            [MethodImpl(MethodImplOptions.AggressiveInlining)]
22            protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref _links[link];
23
24            [MethodImpl(MethodImplOptions.AggressiveInlining)]
25            protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
26        }
27    }
```

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

```
1   using System;
2   using System.Collections.Generic;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences
7   {
8       public static class ArrayExtensions
9       {
10          public static IList<TLink> ConvertToRestrictionsValues<TLink>(this TLink[] array)
11          {
12              var restrictions = new TLink[array.Length + 1];
13              Array.Copy(array, 0, restrictions, 1, array.Length);
14              return restrictions;
15          }
16      }
17  }
```

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

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

```csharp
                    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

```csharp
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;

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

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

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

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

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

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

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

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

        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
            minFrequencyToCompress, bool doInitialFrequenciesIncrement)
            : base(links)
        {
```

```
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      }

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

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)
77      {
78          if (sequence.IsNullOrEmpty())
79          {
80              return null;
81          }
82          if (sequence.Count == 1)
83          {
84              return sequence;
85          }
86          if (sequence.Count == 2)
87          {
88              return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
89          }
90          // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
91          var copy = new HalfDoublet[sequence.Count];
92          Doublet<TLink> doublet = default;
93          for (var i = 1; i < sequence.Count; i++)
94          {
95              doublet.Source = sequence[i - 1];
96              doublet.Target = sequence[i];
97              LinkFrequency<TLink> data;
98              if (_doInitialFrequenciesIncrement)
99              {
100                 data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
101             }
102             else
103             {
104                 data = _doubletFrequenciesCache.GetFrequency(ref doublet);
105                 if (data == null)
106                 {
107                     throw new NotSupportedException("If you ask not to increment
↪                       frequencies, it is expected that all frequencies for the sequence
↪                       are prepared.");
108                 }
109             }
110             copy[i - 1].Element = sequence[i - 1];
111             copy[i - 1].DoubletData = data;
112             UpdateMaxDoublet(ref doublet, data);
113         }
114         copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
115         copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
116         if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
117         {
118             var newLength = ReplaceDoublets(copy);
119             sequence = new TLink[newLength];
120             for (int i = 0; i < newLength; i++)
121             {
122                 sequence[i] = copy[i].Element;
123             }
124         }
125         return sequence;
126     }

128     /// <remarks>
129     /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
130     /// </remarks>
131     private int ReplaceDoublets(HalfDoublet[] copy)
132     {
133         var oldLength = copy.Length;
134         var newLength = copy.Length;
```

```csharp
            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)
                        {
                            var previous = copy[w - 1].Element;
                            copy[w - 1].DoubletData.DecrementFrequency();
                            copy[w - 1].DoubletData =
                            ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
                            ↪ maxDoubletReplacementLink);
                        }
                        if (r < oldLengthMinusTwo)
                        {
                            var next = copy[r + 2].Element;
                            copy[r + 1].DoubletData.DecrementFrequency();
                            copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma⌋
                            ↪ xDoubletReplacementLink,
                            ↪ next);
                        }
                        copy[w++].Element = maxDoubletReplacementLink;
                        r++;
                        newLength--;
                    }
                    else
                    {
                        copy[w++] = copy[r];
                    }
                }
                if (w < newLength)
                {
                    copy[w] = copy[r];
                }
                oldLength = newLength;
                ResetMaxDoublet();
                UpdateMaxDoublet(copy, newLength);
            }
            return newLength;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ResetMaxDoublet()
        {
            _maxDoublet = new Doublet<TLink>();
            _maxDoubletData = new LinkFrequency<TLink>();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
        {
            Doublet<TLink> doublet = default;
            for (var i = 1; i < length; i++)
            {
                doublet.Source = copy[i - 1].Element;
                doublet.Target = copy[i].Element;
                UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
        {
            var frequency = data.Frequency;
            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

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

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

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

```
40                    for (var i = 1; i < length; i++)
41                    {
42                        if (_equalityComparer.Equals(currentLevel, levels[i]))
43                        {
44                            levelRepeat++;
45                            skipOnce = false;
46                            if (levelRepeat == 2)
47                            {
48                                sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
49                                var newLevel = i >= length - 1 ?
50                                    GetPreviousLowerThanCurrentOrCurrent(previousLevel,
                                    ↪ currentLevel) :
51                                    i < 2 ?
52                                    GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
53                                    GetGreatestNeigbourLowerThanCurrentOrCurrent(previousLevel,
                                    ↪ currentLevel, levels[i + 1]);
54                                levels[w] = newLevel;
55                                previousLevel = currentLevel;
56                                w++;
57                                levelRepeat = 0;
58                                skipOnce = true;
59                            }
60                            else if (i == length - 1)
61                            {
62                                sequence[w] = sequence[i];
63                                levels[w] = levels[i];
64                                w++;
65                            }
66                        }
67                        else
68                        {
69                            currentLevel = levels[i];
70                            levelRepeat = 1;
71                            if (skipOnce)
72                            {
73                                skipOnce = false;
74                            }
75                            else
76                            {
77                                sequence[w] = sequence[i - 1];
78                                levels[w] = levels[i - 1];
79                                previousLevel = levels[w];
80                                w++;
81                            }
82                            if (i == length - 1)
83                            {
84                                sequence[w] = sequence[i];
85                                levels[w] = levels[i];
86                                w++;
87                            }
88                        }
89                    }
90                    length = w;
91                }
92                return links.GetOrCreate(sequence[0], sequence[1]);
93            }
94
95        private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink
            ↪ current, TLink next)
96        {
97            return _comparer.Compare(previous, next) > 0
98                ? _comparer.Compare(previous, current) < 0 ? previous : current
99                : _comparer.Compare(next, current) < 0 ? next : current;
100       }
101
102       private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
            ↪ _comparer.Compare(next, current) < 0 ? next : current;
103
104       private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
            ↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
105   }
106 }
```

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

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

```
6  namespace Platform.Data.Doublets.Sequences.Converters
7  {
8      public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
       ↪  IConverter<IList<TLink>>
9      {
10         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11
12         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
13
14         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
       ↪  IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
       ↪  => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
15
16         public IList<TLink> Convert(IList<TLink> sequence)
17         {
18             var levels = new TLink[sequence.Count];
19             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
20             for (var i = 1; i < sequence.Count - 1; i++)
21             {
22                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
23                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
24                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
25             }
26             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
       ↪  sequence[sequence.Count - 1]);
27             return levels;
28         }
29
30         public TLink GetFrequencyNumber(TLink source, TLink target) =>
       ↪  _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
31     }
32  }
```

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

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

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

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

```
1   using System.Collections.Generic;
2   using Platform.Collections.Stacks;
3   using Platform.Data.Doublets.Sequences.HeightProviders;
4   using Platform.Data.Sequences;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences
9   {
10      public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
        ↪  ISequenceAppender<TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
13
14          private readonly IStack<TLink> _stack;
15          private readonly ISequenceHeightProvider<TLink> _heightProvider;
16
17          public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
            ↪  ISequenceHeightProvider<TLink> heightProvider)
18              : base(links)
19          {
20              _stack = stack;
21              _heightProvider = heightProvider;
22          }
23
24          public TLink Append(TLink sequence, TLink appendant)
25          {
26              var cursor = sequence;
27              while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
28              {
29                  var source = Links.GetSource(cursor);
30                  var target = Links.GetTarget(cursor);
31                  if (_equalityComparer.Equals(_heightProvider.Get(source),
                    ↪  _heightProvider.Get(target)))
32                  {
33                      break;
34                  }
35                  else
36                  {
37                      _stack.Push(source);
38                      cursor = target;
39                  }
40              }
41              var left = cursor;
42              var right = appendant;
43              while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
44              {
45                  right = Links.GetOrCreate(left, right);
46                  left = cursor;
47              }
48              return Links.GetOrCreate(left, right);
49          }
50      }
51  }
```

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

```
1   using System;
2   using System.Linq;
```

```csharp
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;

        private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
            IList<TLink>>>
        {
            private readonly IListEqualityComparer<TLink> _listComparer;
            public ItemEquilityComparer() => _listComparer =
                Default<IListEqualityComparer<TLink>>.Instance;
            public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
                KeyValuePair<IList<TLink>, IList<TLink>> right) =>
                _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
                right.Value);
            public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
                (_listComparer.GetHashCode(pair.Key),
                _listComparer.GetHashCode(pair.Value)).GetHashCode();
        }

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

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

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

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

        public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
        {
            _groups = new HashSet<KeyValuePair<IList<TLink>,
                IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
            var count = _links.Count();
            _visited = new BitString((long)(Integer<TLink>)count + 1);
            _links.Each(link =>
            {
                var linkIndex = _links.GetIndex(link);
                var linkBitIndex = (long)(Integer<TLink>)linkIndex;
                if (!_visited.Get(linkBitIndex))
                {
                    var sequenceElements = new List<TLink>();
                    var filler = new ListFiller<TLink, TLink>(sequenceElements,
                        _sequences.Constants.Break);
                    _sequences.Each(filler.AddAllValuesAndReturnConstant, new
                        LinkAddress<TLink>(linkIndex));
```

```csharp
                        if (sequenceElements.Count > 2)
                        {
                            WalkAll(sequenceElements);
                        }
                    }
                    return _links.Constants.Continue;
                });
                var resultList = _groups.ToList();
                var comparer = Default<ItemComparer>.Instance;
                resultList.Sort(comparer);
#if DEBUG
                foreach (var item in resultList)
                {
                    PrintDuplicates(item);
                }
#endif
                return resultList;
            }

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

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

            private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
            {
                var duplicates = new List<TLink>();
                var readAsElement = new HashSet<TLink>();
                var restrictions = segment.ConvertToRestrictionsValues();
                restrictions[0] = _sequences.Constants.Any;
                _sequences.Each(sequence =>
                {
                    var sequenceIndex = sequence[_sequences.Constants.IndexPart];
                    duplicates.Add(sequenceIndex);
                    readAsElement.Add(sequenceIndex);
                    return _sequences.Constants.Continue;
                }, restrictions);
                if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
                {
                    return new List<TLink>();
                }
                foreach (var duplicate in duplicates)
                {
                    var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
                    _visited.Set(duplicateBitIndex);
                }
                if (_sequences is Sequences sequencesExperiments)
                {
                    var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H↲
                    ↪  ashSet<ulong>)(object)readAsElement,
                    ↪  (IList<ulong>)segment);
                    foreach (var partiallyMatchedSequence in partiallyMatched)
                    {
                        TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
                        duplicates.Add(sequenceIndex);
                    }
                }
                duplicates.Sort();
                return duplicates;
            }

            private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
            {
                if (!(_links is ILinks<ulong> ulongLinks))
                {
                    return;
                }
                var duplicatesKey = duplicatesItem.Key;
                var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
                Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
```

```
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;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
10 {
11     /// <remarks>
12     /// Can be used to operate with many CompressingConverters (to keep global frequencies data
        ↪   between them).
13     /// TODO: Extract interface to implement frequencies storage inside Links storage
14     /// </remarks>
15     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪   EqualityComparer<TLink>.Default;
18         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19
20         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
21         private readonly ICounter<TLink, TLink> _frequencyCounter;
22
23         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
24             : base(links)
25         {
26             _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
        ↪   DoubletComparer<TLink>.Default);
27             _frequencyCounter = frequencyCounter;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
32         {
33             var doublet = new Doublet<TLink>(source, target);
34             return GetFrequency(ref doublet);
35         }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
39         {
40             _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
41             return data;
42         }
43
44         public void IncrementFrequencies(IList<TLink> sequence)
45         {
46             for (var i = 1; i < sequence.Count; i++)
47             {
48                 IncrementFrequency(sequence[i - 1], sequence[i]);
49             }
50         }
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
54         {
55             var doublet = new Doublet<TLink>(source, target);
56             return IncrementFrequency(ref doublet);
57         }
```

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

```csharp
#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

```csharp
using Platform.Interfaces;

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

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

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

```csharp
using Platform.Interfaces;

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

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

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

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

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

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

```csharp
#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

```csharp
using Platform.Interfaces;

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

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

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

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

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

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

```csharp
#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

```csharp
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class 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

```csharp
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)
        {
```

```
39          var any = _links.Constants.Any;
40          if (_equalityComparer.Equals(_links.Count(any, link), default))
41          {
42              CountSequenceSymbolFrequency(link);
43          }
44          else
45          {
46              _links.Each(EachElementHandler, any, link);
47          }
48      }
49
50      protected virtual void CountSequenceSymbolFrequency(TLink link)
51      {
52          var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
            ↪  link, _symbol);
53          _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
54      }
55
56      private TLink EachElementHandler(IList<TLink> doublet)
57      {
58          var constants = _links.Constants;
59          var doubletIndex = doublet[constants.IndexPart];
60          if (_visits.Add(doubletIndex))
61          {
62              CountCore(doubletIndex);
63          }
64          return constants.Continue;
65      }
66  }
67 }
```

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

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

```
47              }
48              return height;
49          }
50      }
51  }
```

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

```csharp
1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.HeightProviders
7  {
8      public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
       ↪  ISequenceHeightProvider<TLink>
9      {
10          private readonly ICriterionMatcher<TLink> _elementMatcher;
11
12          public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
            ↪  elementMatcher) : base(links) => _elementMatcher = elementMatcher;
13
14          public TLink Get(TLink sequence)
15          {
16              var height = default(TLink);
17              var pairOrElement = sequence;
18              while (!_elementMatcher.IsMatched(pairOrElement))
19              {
20                  pairOrElement = Links.GetTarget(pairOrElement);
21                  height = Arithmetic.Increment(height);
22              }
23              return height;
24          }
25      }
26  }
```

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

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

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

```csharp
1  using Platform.Collections;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public static class IListExtensions
9      {
10          public static TLink[] ExtractValues<TLink>(this IList<TLink> restrictions)
11          {
12              if(restrictions.IsNullOrEmpty() || restrictions.Count == 1)
13              {
14                  return new TLink[0];
15              }
16              var values = new TLink[restrictions.Count - 1];
17              for (int i = 1, j = 0; i < restrictions.Count; i++, j++)
18              {
19                  values[j] = restrictions[i];
20              }
21              return values;
22          }
23
24          public static IList<TLink> ConvertToRestrictionsValues<TLink>(this IList<TLink> list)
25          {
26              var restrictions = new TLink[list.Count + 1];
27              for (int i = 0, j = 1; i < list.Count; i++, j++)
28              {
29                  restrictions[j] = list[i];
30              }
```

```
31          return restrictions;
32      }
33    }
34  }
```

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

```
1   using System.Collections.Generic;
2   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.Indexes
7   {
8       public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
9       {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
11
12          private readonly LinkFrequenciesCache<TLink> _cache;
13
14          public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
            ↪ _cache = cache;
15
16          public bool Add(IList<TLink> sequence)
17          {
18              var indexed = true;
19              var i = sequence.Count;
20              while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
                ↪ { }
21              for (; i >= 1; i--)
22              {
23                  _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
24              }
25              return indexed;
26          }
27
28          private bool IsIndexedWithIncrement(TLink source, TLink target)
29          {
30              var frequency = _cache.GetFrequency(source, target);
31              if (frequency == null)
32              {
33                  return false;
34              }
35              var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
36              if (indexed)
37              {
38                  _cache.IncrementFrequency(source, target);
39              }
40              return indexed;
41          }
42
43          public bool MightContain(IList<TLink> sequence)
44          {
45              var indexed = true;
46              var i = sequence.Count;
47              while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
48              return indexed;
49          }
50
51          private bool IsIndexed(TLink source, TLink target)
52          {
53              var frequency = _cache.GetFrequency(source, target);
54              if (frequency == null)
55              {
56                  return false;
57              }
58              return !_equalityComparer.Equals(frequency.Frequency, default);
59          }
60      }
61  }
```

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

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

```csharp
        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

```csharp
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

```csharp
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>
    {
```

```
 9              private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪   EqualityComparer<TLink>.Default;
10
11              public SequenceIndex(ILinks<TLink> links) : base(links) { }
12
13              public virtual bool Add(IList<TLink> sequence)
14              {
15                  var indexed = true;
16                  var i = sequence.Count;
17                  while (--i >= 1 && (indexed =
        ↪   !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
        ↪   default))) { }
18                  for (; i >= 1; i--)
19                  {
20                      Links.GetOrCreate(sequence[i - 1], sequence[i]);
21                  }
22                  return indexed;
23              }
24
25              public virtual bool MightContain(IList<TLink> sequence)
26              {
27                  var indexed = true;
28                  var i = sequence.Count;
29                  while (--i >= 1 && (indexed =
        ↪   !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
        ↪   default))) { }
30                  return indexed;
31              }
32          }
33  }
```

./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs
```
 1  using System.Collections.Generic;
 2
 3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 4
 5  namespace Platform.Data.Doublets.Sequences.Indexes
 6  {
 7      public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
 8      {
 9          private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪   EqualityComparer<TLink>.Default;
10
11          private readonly ISynchronizedLinks<TLink> _links;
12
13          public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
14
15          public bool Add(IList<TLink> sequence)
16          {
17              var indexed = true;
18              var i = sequence.Count;
19              var links = _links.Unsync;
20              _links.SyncRoot.ExecuteReadOperation(() =>
21              {
22                  while (--i >= 1 && (indexed =
            ↪   !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
            ↪   sequence[i]), default))) { }
23              });
24              if (!indexed)
25              {
26                  _links.SyncRoot.ExecuteWriteOperation(() =>
27                  {
28                      for (; i >= 1; i--)
29                      {
30                          links.GetOrCreate(sequence[i - 1], sequence[i]);
31                      }
32                  });
33              }
34              return indexed;
35          }
36
37          public bool MightContain(IList<TLink> sequence)
38          {
39              var links = _links.Unsync;
40              return _links.SyncRoot.ExecuteReadOperation(() =>
41              {
42                  var indexed = true;
43                  var i = sequence.Count;
```

```
44            while (--i >= 1 && (indexed =
    ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
    ↪  sequence[i]), default))) { }
45            return indexed;
46         });
47      }
48   }
49  }
```

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

```
1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Indexes
6  {
7      public class Unindex<TLink> : ISequenceIndex<TLink>
8      {
9          public virtual bool Add(IList<TLink> sequence) => false;
10
11          public virtual bool MightContain(IList<TLink> sequence) => true;
12      }
13  }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public class ListFiller<TElement, TReturnConstant>
9      {
10          protected readonly List<TElement> _list;
11          protected readonly TReturnConstant _returnConstant;
12
13          public ListFiller(List<TElement> list, TReturnConstant returnConstant)
14          {
15              _list = list;
16              _returnConstant = returnConstant;
17          }
18
19          public ListFiller(List<TElement> list) : this(list, default) { }
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          public void Add(TElement element) => _list.Add(element);
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          public bool AddAndReturnTrue(TElement element)
26          {
27              _list.Add(element);
28              return true;
29          }
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public bool AddFirstAndReturnTrue(IList<TElement> collection)
33          {
34              _list.Add(collection[0]);
35              return true;
36          }
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          public TReturnConstant AddAndReturnConstant(TElement element)
40          {
41              _list.Add(element);
42              return _returnConstant;
43          }
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          public TReturnConstant AddFirstAndReturnConstant(IList<TElement> collection)
47          {
48              _list.Add(collection[0]);
49              return _returnConstant;
50          }
51
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
53          public TReturnConstant AddAllValuesAndReturnConstant(IList<TElement> collection)
54          {
55              for (int i = 1; i < collection.Count; i++)
```

```csharp
            {
                _list.Add(collection[i]);
            }
            return _returnConstant;
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// Представляет коллекцию последовательностей связей.
    /// </summary>
    /// <remarks>
    /// Обязательно реализовать атомарность каждого публичного метода.
    ///
    /// TODO:
    ///
    /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
    /// через естественную группировку по unicode типам, все whitespace вместе, все символы
    ///   вместе, все числа вместе и т.п.
    /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
    ///   графа)
    ///
    /// x*y - найти все связи между, в последовательностях любой формы, если не стоит
    ///   ограничитель на то, что является последовательностью, а что нет,
    /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
    ///   порядке.
    ///
    /// Рост последовательности слева и справа.
    /// Поиск со звёздочкой.
    /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
    /// так же проблема может быть решена при реализации дистанционных триггеров.
    /// Нужны ли уникальные указатели вообще?
    /// Что если обращение к информации будет происходить через содержимое всегда?
    ///
    /// Писать тесты.
    ///
    ///
    /// Можно убрать зависимость от конкретной реализации Links,
    /// на зависимость от абстрактного элемента, который может быть представлен несколькими
    ///   способами.
    ///
    /// Можно ли как-то сделать один общий интерфейс
    ///
    ///
    /// Блокчейн и/или гит для распределённой записи транзакций.
    ///
    /// </remarks>
    public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
      (после завершения реализации Sequences)
    {
        /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
        ///   связей.</summary>
        public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;

        public SequencesOptions<LinkIndex> Options { get; }
        public SynchronizedLinks<LinkIndex> Links { get; }
        private readonly ISynchronization _sync;

        public LinksConstants<LinkIndex> Constants { get; }

        public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
        {
```

```csharp
                    Links = links;
                    _sync = links.SyncRoot;
                    Options = options;
                    Options.ValidateOptions();
                    Options.InitOptions(Links);
                    Constants = links.Constants;
            }

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

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

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

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

            #region Count

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

```csharp
143
144          private LinkIndex CountUsages(params LinkIndex[] restrictions)
145          {
146              if (restrictions.Length == 0)
147              {
148                  return 0;
149              }
150              if (restrictions.Length == 1) // Первая связь это адрес
151              {
152                  if (restrictions[0] == Constants.Null)
153                  {
154                      return 0;
155                  }
156                  if (Options.UseSequenceMarker)
157                  {
158                      var elementsLink = GetSequenceElements(restrictions[0]);
159                      var sequenceLink = GetSequenceByElements(elementsLink);
160                      if (sequenceLink != Constants.Null)
161                      {
162                          return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
163                      }
164                      return Links.Count(elementsLink);
165                  }
166                  return Links.Count(restrictions[0]);
167              }
168              throw new NotImplementedException();
169          }
170
171          #endregion
172
173          #region Create
174
175          public LinkIndex Create(IList<LinkIndex> restrictions)
176          {
177              return _sync.ExecuteWriteOperation(() =>
178              {
179                  if (restrictions.IsNullOrEmpty())
180                  {
181                      return Constants.Null;
182                  }
183                  Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
184                  return CreateCore(restrictions);
185              });
186          }
187
188          private LinkIndex CreateCore(IList<LinkIndex> restrictions)
189          {
190              LinkIndex[] sequence = restrictions.ExtractValues();
191              if (Options.UseIndex)
192              {
193                  Options.Index.Add(sequence);
194              }
195              var sequenceRoot = default(LinkIndex);
196              if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
197              {
198                  var matches = Each(restrictions);
199                  if (matches.Count > 0)
200                  {
201                      sequenceRoot = matches[0];
202                  }
203              }
204              else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
205              {
206                  return CompactCore(sequence);
207              }
208              if (sequenceRoot == default)
209              {
210                  sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
211              }
212              if (Options.UseSequenceMarker)
213              {
214                  Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
215              }
216              return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
217          }
218
219          #endregion
220
221          #region Each
```

```csharp
222
223        public List<LinkIndex> Each(IList<LinkIndex> sequence)
224        {
225            var results = new List<LinkIndex>();
226            var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
227            Each(filler.AddFirstAndReturnConstant, sequence);
228            return results;
229        }
230
231        public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↪ restrictions)
232        {
233            return _sync.ExecuteReadOperation(() =>
234            {
235                if (restrictions.IsNullOrEmpty())
236                {
237                    return Constants.Continue;
238                }
239                Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
240                if (restrictions.Count == 1)
241                {
242                    var link = restrictions[0];
243                    var any = Constants.Any;
244                    if (link == any)
245                    {
246                        if (Options.UseSequenceMarker)
247                        {
248                            return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
                                ↪ Options.SequenceMarkerLink, any));
249                        }
250                        else
251                        {
252                            return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
                                ↪ any));
253                        }
254                    }
255                    if (Options.UseSequenceMarker)
256                    {
257                        var sequenceLinkValues = Links.Unsync.GetLink(link);
258                        if (sequenceLinkValues[Constants.SourcePart] ==
                            ↪ Options.SequenceMarkerLink)
259                        {
260                            link = sequenceLinkValues[Constants.TargetPart];
261                        }
262                    }
263                    var sequence =
                        ↪ Options.Walker.Walk(link).ToArray().ConvertToRestrictionsValues();
264                    sequence[0] = link;
265                    return handler(sequence);
266                }
267                else if (restrictions.Count == 2)
268                {
269                    throw new NotImplementedException();
270                }
271                else if (restrictions.Count == 3)
272                {
273                    return Links.Unsync.Each(handler, restrictions);
274                }
275                else
276                {
277                    var sequence = restrictions.ExtractValues();
278                    if (Options.UseIndex && !Options.Index.MightContain(sequence))
279                    {
280                        return Constants.Break;
281                    }
282                    return EachCore(handler, sequence);
283                }
284            });
285        }
286
287        private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↪ values)
288        {
289            var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
290            // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
                ↪ Id.
```

```csharp
            Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
                (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
                matcher.HandleFullMatched;
            //if (sequence.Length >= 2)
            if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
            {
                return Constants.Break;
            }
            var last = values.Count - 2;
            for (var i = 1; i < last; i++)
            {
                if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
                    Constants.Continue)
                {
                    return Constants.Break;
                }
            }
            if (values.Count >= 3)
            {
                if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
                    != Constants.Continue)
                {
                    return Constants.Break;
                }
            }
            return Constants.Continue;
        }

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

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

        private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            right, LinkIndex stepFrom)
        {
            var upStep = stepFrom;
            var firstSource = Links.Unsync.GetTarget(upStep);
            while (firstSource != right && firstSource != upStep)
            {
                upStep = firstSource;
                firstSource = Links.Unsync.GetSource(upStep);
            }
            if (firstSource == right)
            {
                return handler(new LinkAddress<LinkIndex>(stepFrom));
            }
            return Constants.Continue;
        }

        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));

        private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            left, LinkIndex stepFrom)
        {
            var upStep = stepFrom;
            var firstTarget = Links.Unsync.GetSource(upStep);
```

```csharp
                    while (firstTarget != left && firstTarget != upStep)
                    {
                        upStep = firstTarget;
                        firstTarget = Links.Unsync.GetTarget(upStep);
                    }
                    if (firstTarget == left)
                    {
                        return handler(new LinkAddress<LinkIndex>(stepFrom));
                    }
                    return Constants.Continue;
                }

        #endregion

        #region Update

        public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
        {
            var sequence = restrictions.ExtractValues();
            var newSequence = substitution.ExtractValues();

            if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
            {
                return Constants.Null;
            }
            if (sequence.IsNullOrEmpty())
            {
                return Create(substitution);
            }
            if (newSequence.IsNullOrEmpty())
            {
                Delete(restrictions);
                return Constants.Null;
            }
            return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
            {
                ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
                Links.EnsureLinkExists(newSequence);
                return UpdateCore(sequence, newSequence);
            }));
        }

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

        private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(sequence);
                var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                var newSequenceElements = GetSequenceElements(newSequence);
                var newSequenceLink = GetSequenceByElements(newSequenceElements);
                if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
```

```csharp
                {
                    if (sequenceLink != Constants.Null)
                    {
                        Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
                    }
                    Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(sequence);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    var newSequenceElements = GetSequenceElements(newSequence);
                    var newSequenceLink = GetSequenceByElements(newSequenceElements);
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        if (sequenceLink != Constants.Null)
                        {
                            Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
                        }
                        Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
                    }
                }
                else
                {
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        Links.Unsync.MergeUsages(sequence, newSequence);
                    }
                }
            }
        }
    }

    #endregion

    #region Delete

    public void Delete(IList<LinkIndex> restrictions)
    {
        _sync.ExecuteWriteOperation(() =>
        {
            var sequence = restrictions.ExtractValues();
            // TODO: Check all options only ones before loop execution
            foreach (var linkToDelete in Each(sequence))
            {
                DeleteOneCore(linkToDelete);
            }
        });
    }

    private void DeleteOneCore(LinkIndex link)
    {
        if (Options.UseGarbageCollection)
        {
            var sequenceElements = GetSequenceElements(link);
            var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
            var sequenceLink = GetSequenceByElements(sequenceElements);
            if (Options.UseCascadeDelete || CountUsages(link) == 0)
            {
                if (sequenceLink != Constants.Null)
                {
                    Links.Unsync.Delete(sequenceLink);
                }
                Links.Unsync.Delete(link);
            }
            ClearGarbage(sequenceElementsContents.Source);
            ClearGarbage(sequenceElementsContents.Target);
        }
        else
        {
            if (Options.UseSequenceMarker)
            {
                var sequenceElements = GetSequenceElements(link);
                var sequenceLink = GetSequenceByElements(sequenceElements);
```

```csharp
510                        if (Options.UseCascadeDelete || CountUsages(link) == 0)
511                        {
512                            if (sequenceLink != Constants.Null)
513                            {
514                                Links.Unsync.Delete(sequenceLink);
515                            }
516                            Links.Unsync.Delete(link);
517                        }
518                    }
519                    else
520                    {
521                        if (Options.UseCascadeDelete || CountUsages(link) == 0)
522                        {
523                            Links.Unsync.Delete(link);
524                        }
525                    }
526                }
527            }

528
529            #endregion

530
531            #region Compactification

532
533            public void CompactAll()
534            {
535                _sync.ExecuteWriteOperation(() =>
536                {
537                    var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
538                    for (int i = 0; i < sequences.Count; i++)
539                    {
540                        var sequence = this.ToList(sequences[i]);
541                        Compact(sequence.ConvertToRestrictionsValues());
542                    }
543                });
544            }

545
546            /// <remarks>
547            /// bestVariant можно выбирать по максимальному числу использований,
548            /// но балансированный позволяет гарантировать уникальность (если есть возможность,
549            /// гарантировать его использование в других местах).
550            /// ///
551            /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
552            /// </remarks>
553            public LinkIndex Compact(IList<LinkIndex> sequence)
554            {
555                return _sync.ExecuteWriteOperation(() =>
556                {
557                    if (sequence.IsNullOrEmpty())
558                    {
559                        return Constants.Null;
560                    }
561                    Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
562                    return CompactCore(sequence);
563                });
564            }

565
566            [MethodImpl(MethodImplOptions.AggressiveInlining)]
567            private LinkIndex CompactCore(IList<LinkIndex> sequence) => UpdateCore(sequence,
      ↪ sequence);

568
569            #endregion

570
571            #region Garbage Collection

572
573            /// <remarks>
574            /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
      ↪ определить извне или в унаследованном классе
575            /// </remarks>
576            [MethodImpl(MethodImplOptions.AggressiveInlining)]
577            private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
      ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;

578
579            private void ClearGarbage(LinkIndex link)
580            {
581                if (IsGarbage(link))
582                {
583                    var contents = new Link<ulong>(Links.GetLink(link));
584                    Links.Unsync.Delete(link);
585                    ClearGarbage(contents.Source);
```

```csharp
                    ClearGarbage(contents.Target);
                }
        }

        #endregion

        #region Walkers

        public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var links = Links.Unsync;
                foreach (var part in Options.Walker.Walk(sequence))
                {
                    if (!handler(part))
                    {
                        return false;
                    }
                }
                return true;
            });
        }

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

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

            protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
                (_readAsElements != null && _readAsElements.Contains(link)) ||
                _linksInSequence.Contains(link);

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

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

```csharp
                }

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

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

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

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

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

        public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
        {
            if (PartialMatch(sequenceToMatch))
```

```csharp
                        {
                            _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

```csharp
using System;
using LinkIndex = System.UInt64;
using System.Collections.Generic;
using Stack = System.Collections.Generic.Stack<ulong>;
using System.Linq;
using System.Text;
using Platform.Collections;
using Platform.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.EnsureLinkExists(sequence);
                if (sequence.Length == 1)
```

```csharp
                {
                    return sequence;
                }
                return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
            });
        }

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

        public List<ulong> CreateAllVariants1(params ulong[] sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return new List<ulong>();
                }
                Links.Unsync.EnsureLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return new List<ulong> { sequence[0] };
                }
                var results = new
                ↪  List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
                return CreateAllVariants1Core(sequence, results);
            });
        }

        private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
        {
            if (sequence.Length == 2)
            {
                var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
                if (link == Constants.Null)
                {
                    throw new NotImplementedException("Creation cancellation is not
                    ↪  implemented.");
                }
                results.Add(link);
```

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

            #endregion

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

            private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
            {
                if (sequence.Length == 2)
                {
                    Links.Unsync.Each(sequence[0], sequence[1], handler);
                }
                else
                {
                    var innerSequenceLength = sequence.Length - 1;
                    for (var li = 0; li < innerSequenceLength; li++)
                    {
                        var left = sequence[li];
                        var right = sequence[li + 1];
                        if (left == 0 && right == 0)
                        {
                            continue;
                        }
                        var linkIndex = li;
                        ulong[] innerSequence = null;
                        Links.Unsync.Each(doublet =>
                        {
                            if (innerSequence == null)
                            {
                                innerSequence = new ulong[innerSequenceLength];
                                for (var isi = 0; isi < linkIndex; isi++)
                                {
                                    innerSequence[isi] = sequence[isi];
                                }
                                for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
                                {
                                    innerSequence[isi] = sequence[isi + 1];
                                }
                            }
                            innerSequence[linkIndex] = doublet[Constants.IndexPart];
                            Each1(handler, innerSequence);
                            return Constants.Continue;
                        }, Constants.Any, left, right);
```

```csharp
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.EnsureLinkIsAnyOrExists(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
236                    {
237                        Links.Each(Constants.Any, Constants.Any, handler);
238                    }
239                }
240                else if (sequence.Length == 2)
241                {
242                    //_links.Each(sequence[0], sequence[1], handler);
243                    //  o_|        x_o ...
244                    //  x_|        |___|
245                    Links.Each(sequence[1], Constants.Any, doublet =>
246                    {
247                        var match = Links.SearchOrDefault(sequence[0], doublet);
248                        if (match != Constants.Null)
249                        {
250                            handler(new LinkAddress<LinkIndex>(match));
251                        }
252                        return true;
253                    });
254                    //  |_x        ... x_o
255                    //  |_o        |___|
256                    Links.Each(Constants.Any, sequence[0], doublet =>
257                    {
258                        var match = Links.SearchOrDefault(doublet, sequence[1]);
259                        if (match != 0)
260                        {
261                            handler(new LinkAddress<LinkIndex>(match));
262                        }
263                        return true;
```

```
264              });
265              //              ._x o_.
266              //               |___|
267              PartialStepRight(x => handler(x), sequence[0], sequence[1]);
268          }
269          else
270          {
271              throw new NotImplementedException();
272          }
273      }
274
275      private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
276      {
277          Links.Unsync.Each(Constants.Any, left, doublet =>
278          {
279              StepRight(handler, doublet, right);
280              if (left != doublet)
281              {
282                  PartialStepRight(handler, doublet, right);
283              }
284              return true;
285          });
286      }
287
288      private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
289      {
290          Links.Unsync.Each(left, Constants.Any, rightStep =>
291          {
292              TryStepRightUp(handler, right, rightStep);
293              return true;
294          });
295      }
296
297      private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
298      ↪  stepFrom)
299      {
299          var upStep = stepFrom;
300          var firstSource = Links.Unsync.GetTarget(upStep);
301          while (firstSource != right && firstSource != upStep)
302          {
303              upStep = firstSource;
304              firstSource = Links.Unsync.GetSource(upStep);
305          }
306          if (firstSource == right)
307          {
308              handler(new LinkAddress<LinkIndex>(stepFrom));
309          }
310      }
311
312      // 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.EnsureLinkExists(sequence);
382                    var firstElement = sequence[0];
383                    if (sequence.Length == 1)
384                    {
385                        results.Add(firstElement);
386                        return results;
387                    }
388                    if (sequence.Length == 2)
389                    {
390                        var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
391                        if (doublet != Constants.Null)
392                        {
393                            results.Add(doublet);
394                        }
395                        return results;
396                    }
397                    var linksInSequence = new HashSet<ulong>(sequence);
398                    void handler(IList<LinkIndex> result)
399                    {
400                        var resultIndex = result[Links.Constants.IndexPart];
401                        var filterPosition = 0;
402                        StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
                        ↪  Links.Unsync.GetTarget,
403                            x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                            ↪  x =>
404                            {
405                                if (filterPosition == sequence.Length)
406                                {
407                                    filterPosition = -2; // Длиннее чем нужно
408                                    return false;
409                                }
410                                if (x != sequence[filterPosition])
411                                {
412                                    filterPosition = -1;
413                                    return false; // Начинается иначе
414                                }
415                                filterPosition++;
416
417                                return true;
418                            });
```

```csharp
                            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.EnsureLinkExists(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));
```

```
490        private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
      ↪    Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
      ↪    LinkIndex[] knownElements)
491        {
492            var linksInSequence = new HashSet<ulong>(knownElements);
493            //var entered = new HashSet<ulong>();
494            var sb = new StringBuilder();
495            sb.Append('{');
496            if (links.Exists(sequenceLink))
497            {
498                StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
499                    x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
                  ↪    entered.AddAndReturnVoid, x => { }, entered.DoNotContains
500                    {
501                        if (insertComma && sb.Length > 1)
502                        {
503                            sb.Append(',');
504                        }
505                        //if (entered.Contains(element))
506                        //{
507                        //    sb.Append('{');
508                        //    elementToString(sb, element);
509                        //    sb.Append('}');
510                        //}
511                        //else
512                        elementToString(sb, element);
513                        if (sb.Length < MaxSequenceFormatSize)
514                        {
515                            return true;
516                        }
517                        sb.Append(insertComma ? ", ..." : "...");
518                        return false;
519                    });
520            }
521            sb.Append('}');
522            return sb.ToString();
523        }

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

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

529        private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
      ↪    Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
      ↪    LinkIndex[] knownElements)
530        {
531            var linksInSequence = new HashSet<ulong>(knownElements);
532            var entered = new HashSet<ulong>();
533            var sb = new StringBuilder();
534            sb.Append('{');
535            if (links.Exists(sequenceLink))
536            {
537                StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
538                    x => linksInSequence.Contains(x) || links.IsFullPoint(x),
                  ↪    entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
539                    {
540                        if (insertComma && sb.Length > 1)
541                        {
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                        {
```

```csharp
                                return true;
                            }
                            sb.Append(insertComma ? ", ..." : "...");
                            return false;
                        });
                }
                sb.Append('}');
                return sb.ToString();
            }

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

        public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    var results = new HashSet<ulong>();
                    for (var i = 0; i < sequence.Length; i++)
                    {
                        AllUsagesCore(sequence[i], results);
                    }
```

```
633                         var filteredResults = new HashSet<ulong>();
634                         var matcher = new Matcher(this, sequence, filteredResults, null);
635                         matcher.AddAllPartialMatchedToResults(results);
636                         return filteredResults;
637                     }
638                     return new HashSet<ulong>();
639                 });
640         }

642         public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
        ↪  params ulong[] sequence)
643         {
644             return _sync.ExecuteReadOperation(() =>
645             {
646                 if (sequence.Length > 0)
647                 {
648                     Links.EnsureLinkExists(sequence);

650                     var results = new HashSet<ulong>();
651                     var filteredResults = new HashSet<ulong>();
652                     var matcher = new Matcher(this, sequence, filteredResults, handler);
653                     for (var i = 0; i < sequence.Length; i++)
654                     {
655                         if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
656                         {
657                             return false;
658                         }
659                     }
660                     return true;
661                 }
662                 return true;
663             });
664         }

666         //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
667         //{
668         //     return Sync.ExecuteReadOperation(() =>
669         //     {
670         //         if (sequence.Length > 0)
671         //         {
672         //             _links.EnsureEachLinkIsAnyOrExists(sequence);

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

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

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

683         //             firstResults.IntersectWith(lastResults);

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

688         //             var filteredResults = new HashSet<ulong>();
689         //             var matcher = new Matcher(this, sequence, filteredResults, null);
690         //             matcher.AddAllPartialMatchedToResults(firstResults);
691         //             return filteredResults;
692         //         }

694         //         return new HashSet<ulong>();
695         //     });
696         //}

698         public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
699         {
700             return _sync.ExecuteReadOperation((Func<HashSet<ulong>>)(() =>
701             {
702                 if (sequence.Length > 0)
703                 {
704                     ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
                        ↪  (IList<ulong>)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);
```

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

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

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

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

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

```csharp
                    {
                        if (sequence.Length > 0)
                        {
                            Links.EnsureLinkExists(sequence);
                            //var firstElement = sequence[0];
                            //if (sequence.Length == 1)
                            //{
                            //    //results.Add(firstElement);
                            //    return results;
                            //}
                            //if (sequence.Length == 2)
                            //{
                            //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
                            //    //if (doublet != Doublets.Links.Null)
                            //    //    results.Add(doublet);
                            //    return results;
                            //}
                            //var lastElement = sequence[sequence.Length - 1];
                            //Func<ulong, bool> handler = x =>
                            //{
                            //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
                            //        results.Add(x);
                            //    return true;
                            //};
                            //if (sequence.Length >= 2)
                            //    StepRight(handler, sequence[0], sequence[1]);
                            //var last = sequence.Length - 2;
                            //for (var i = 1; i < last; i++)
                            //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
                            //if (sequence.Length >= 3)
                            //    StepLeft(handler, sequence[sequence.Length - 2],
                            //        sequence[sequence.Length - 1]);
                            //////if (sequence.Length == 1)
                            //////{
                            //////    throw new NotImplementedException(); // all sequences, containing
                            //////    this element?
                            //////}
                            //////if (sequence.Length == 2)
                            //////{
                            //////    var results = new List<ulong>();
                            //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
                            //////    return results;
                            //////}
                            //////var matches = new List<List<ulong>>();
                            //////var last = sequence.Length - 1;
                            //////for (var i = 0; i < last; i++)
                            //////{
                            //////    var results = new List<ulong>();
                            //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
                            //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
                            //////    if (results.Count > 0)
                            //////        matches.Add(results);
                            //////    else
                            //////        return results;
                            //////    if (matches.Count == 2)
                            //////    {
                            //////        var merged = new List<ulong>();
                            //////        for (var j = 0; j < matches[0].Count; j++)
                            //////            for (var k = 0; k < matches[1].Count; k++)
                            //////                CloseInnerConnections(merged.Add, matches[0][j],
                            //////                    matches[1][k]);
                            //////        if (merged.Count > 0)
                            //////            matches = new List<List<ulong>> { merged };
                            //////        else
                            //////            return new List<ulong>();
                            //////    }
                            //////}
                            //////if (matches.Count > 0)
                            //////{
                            //////    var usages = new HashSet<ulong>();
                            //////    for (int i = 0; i < sequence.Length; i++)
                            //////    {
                            //////        AllUsagesCore(sequence[i], usages);
                            //////    }
                            //////    //for (int i = 0; i < matches[0].Count; i++)
                            //////    //    AllUsagesCore(matches[0][i], usages);
                            //////    //usages.UnionWith(matches[0]);
```

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

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

    public HashSet<ulong> AllBottomUsages(ulong link)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            var visits = new HashSet<ulong>();
            var usages = new HashSet<ulong>();
            AllBottomUsagesCore(link, visits, usages);
            return usages;
        });
    }

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

```csharp
                {
                    Links.Unsync.Each(link, Constants.Any, handler);
                    Links.Unsync.Each(Constants.Any, link, handler);
                }
            }

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

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

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

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

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

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

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

            private bool CalculateCore(ulong link)
            {
                if (_totals[link] == 0)
                {
                    var total = 1UL;
                    _totals[link] = total;
                    var visitedChildren = new HashSet<ulong>();
                    bool linkCalculator(ulong child)
                    {
```

```
                             if (link != child && visitedChildren.Add(child))
                             {
                                  total += _totals[child] == 0 ? 1 : _totals[child];
                             }
                             return true;
                        }
                        _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
                        _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
                        _totals[link] = total;
                   }
                   return true;
              }
         }

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

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

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

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

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

```csharp
                    }
                    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))
```

```csharp
                {
                    _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>();
```

```csharp
            var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
            //for (var i = 0; i < previousMatchings.Count; i++)
            foreach (var secondLinkUsage in secondLinkUsages)
            {
                foreach (var previousMatching in previousMatchings)
                {
                    //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
                    ↪ secondLinkUsage);
                    StepRight(filler.AddFirstAndReturnConstant, previousMatching,
                    ↪ secondLinkUsage);
                    TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
                    ↪ previousMatching);
                    //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
                    ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
                    ↪ желаемым результам.
                    PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
                    ↪ secondLinkUsage);
                }
            }
            if (matchings.Count == 0)
            {
                return matchings;
            }
            return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
        }

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

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

        // Найти все возможные связи между указанным списком связей.
```

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

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

            public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
            {
                return _sync.ExecuteReadOperation(() =>
                {
                    var results = new HashSet<ulong>();
                    if (linksToConnect.Length > 0)
                    {
                        Links.EnsureLinkExists(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.EnsureLinkExists(linksToConnect);
                            var collector1 = new AllUsagesCollector2(Links.Unsync, results);
                            collector1.Collect(linksToConnect[0]);
                            for (var i = 1; i < linksToConnect.Length; i++)
                            {
                                var next = new BitString((long)Links.Unsync.Count() + 1); //new
                                ↪  BitArray((int)_links.Total + 1);
                                var collector = new AllUsagesCollector2(Links.Unsync, next);
                                collector.Collect(linksToConnect[i]);
                                results = results.And(next);
                            }
                        }
                        return results.GetSetUInt64Indices();
                    });
                }

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

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

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

```csharp
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                    }
1502                }
1503                else
1504                {
1505                    for (var i = elements.Length - 1; i >= 0; i--)
1506                    {
1507                        var element = elements[i];
1508                        if (element != 0)
1509                        {
1510                            results.Add(element);
1511                        }
1512                    }
1513                }
1514            }
1515            else
1516            {
1517                var nextRightLink = middleLinks[rightBound];
1518                var elements = GetLeftElements(rightLink, nextRightLink);
1519                if (leftBound <= rightBound)
1520                {
1521                    for (var i = elements.Length - 1; i >= 0; i--)
1522                    {
1523                        var element = elements[i];
1524                        if (element != 0)
1525                        {
1526                            CollectMatchingSequences(leftLink, leftBound, middleLinks,
     ↪  elements[i], rightBound - 1, ref results);
1527                        }
1528                    }
1529                }
```

```csharp
                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;
            private readonly HashSet<LinkIndex> _linksInSequence;
            private readonly HashSet<LinkIndex> _results;

            #region Pattern Match

            enum PatternBlockType
            {
                Undefined,
                Gap,
                Elements
            }

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

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

            #endregion

            public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
            ↪    HashSet<LinkIndex> results)
                : base(sequences.Links.Unsync, new DefaultStack<ulong>())
            {
                _sequences = sequences;
                _patternSequence = patternSequence;
```

```csharp
                    _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪  _sequences.Constants.Any && x != ZeroOrMany));
                    _results = results;
                    _pattern = CreateDetailedPattern();
            }

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

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

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

```csharp
                    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;
```

```csharp
            //    }

            //    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;
                //}
                ////////
```

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

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

            #endregion
        }
    }
```

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

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

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

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

        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;
        }
    }
}
```

```
1    using System;
2    using System.Collections.Generic;
3    using Platform.Interfaces;
4    using Platform.Collections.Stacks;
5    using Platform.Data.Doublets.Sequences.Frequencies.Cache;
6    using Platform.Data.Doublets.Sequences.Frequencies.Counters;
7    using Platform.Data.Doublets.Sequences.Converters;
8    using Platform.Data.Doublets.Sequences.CreteriaMatchers;
9    using Platform.Data.Doublets.Sequences.Walkers;
10   using Platform.Data.Doublets.Sequences.Indexes;
11
12   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14   namespace Platform.Data.Doublets.Sequences
15   {
16       public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
         ↪  ILinks<TLink> must contain GetConstants function.
17       {
18           private static readonly EqualityComparer<TLink> _equalityComparer =
             ↪  EqualityComparer<TLink>.Default;
19
20           public TLink SequenceMarkerLink { get; set; }
21           public bool UseCascadeUpdate { get; set; }
22           public bool UseCascadeDelete { get; set; }
23           public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
24           public bool UseSequenceMarker { get; set; }
25           public bool UseCompression { get; set; }
26           public bool UseGarbageCollection { get; set; }
27           public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
28           public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
29
30           public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher { get; set; }
31           public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
32           public ISequenceIndex<TLink> Index { get; set; }
33           public ISequenceWalker<TLink> Walker { get; set; }
34           public bool ReadFullSequence { get; set; }
35
36           // TODO: Реализовать компактификацию при чтении
37           //public bool EnforceSingleSequenceVersionOnRead { get; set; }
38           //public bool UseRequestMarker { get; set; }
39           //public bool StoreRequestResults { get; set; }
40
41           public void InitOptions(ISynchronizedLinks<TLink> links)
42           {
43               if (UseSequenceMarker)
44               {
45                   if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
46                   {
47                       SequenceMarkerLink = links.CreatePoint();
48                   }
49                   else
50                   {
51                       if (!links.Exists(SequenceMarkerLink))
52                       {
53                           var link = links.CreatePoint();
54                           if (!_equalityComparer.Equals(link, SequenceMarkerLink))
55                           {
56                               throw new InvalidOperationException("Cannot recreate sequence marker
                             ↪  link.");
57                           }
58                       }
59                   }
60                   if (MarkedSequenceMatcher == null)
61                   {
62                       MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                     ↪  SequenceMarkerLink);
63                   }
64               }
65               var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
66               if (UseCompression)
67               {
68                   if (LinksToSequenceConverter == null)
69                   {
70                       ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
71                       if (UseSequenceMarker)
72                       {
```

```
73                          totalSequenceSymbolFrequencyCounter = new
                            ↪  TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                            ↪  MarkedSequenceMatcher);
74                      }
75                      else
76                      {
77                          totalSequenceSymbolFrequencyCounter = new
                            ↪  TotalSequenceSymbolFrequencyCounter<TLink>(links);
78                      }
79                      var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                        ↪  totalSequenceSymbolFrequencyCounter);
80                      var compressingConverter = new CompressingConverter<TLink>(links,
                        ↪  balancedVariantConverter, doubletFrequenciesCache);
81                      LinksToSequenceConverter = compressingConverter;
82                  }
83              }
84              else
85              {
86                  if (LinksToSequenceConverter == null)
87                  {
88                      LinksToSequenceConverter = balancedVariantConverter;
89                  }
90              }
91              if (UseIndex && Index == null)
92              {
93                  Index = new SequenceIndex<TLink>(links);
94              }
95              if (Walker == null)
96              {
97                  Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
98              }
99          }

101        public void ValidateOptions()
102        {
103            if (UseGarbageCollection && !UseSequenceMarker)
104            {
105                throw new NotSupportedException("To use garbage collection UseSequenceMarker
                   ↪  option must be on.");
106            }
107        }
108    }
109 }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public class SetFiller<TElement, TReturnConstant>
9      {
10         protected readonly ISet<TElement> _set;
11         protected readonly TReturnConstant _returnConstant;
12
13         public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
14         {
15             _set = set;
16             _returnConstant = returnConstant;
17         }
18
19         public SetFiller(ISet<TElement> set) : this(set, default) { }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public void Add(TElement element) => _set.Add(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public bool AddAndReturnTrue(TElement element)
26         {
27             _set.Add(element);
28             return true;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public bool AddFirstAndReturnTrue(IList<TElement> collection)
33         {
34             _set.Add(collection[0]);
```

```
35              return true;
36          }
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          public TReturnConstant AddAndReturnConstant(TElement element)
40          {
41              _set.Add(element);
42              return _returnConstant;
43          }
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          public TReturnConstant AddFirstAndReturnConstant(IList<TElement> collection)
47          {
48              _set.Add(collection[0]);
49              return _returnConstant;
50          }
51      }
52  }
```

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

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

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Collections.Stacks;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences.Walkers
9   {
10      public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11      {
12          public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
            ↪  isElement) : base(links, stack, isElement) { }
13
14          public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
            ↪  links.IsPartialPoint) { }
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override TLink GetNextElementAfterPop(TLink element) =>
            ↪  Links.GetSource(element);
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override TLink GetNextElementAfterPush(TLink element) =>
            ↪  Links.GetTarget(element);
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected override IEnumerable<TLink> WalkContents(TLink element)
24          {
25              var parts = Links.GetLink(element);
26              var start = Links.Constants.IndexPart + 1;
27              for (var i = parts.Count - 1; i >= start; i--)
28              {
29                  var part = parts[i];
30                  if (IsElement(part))
31                  {
32                      yield return part;
33                  }
34              }
35          }
36      }
37  }
```

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

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

```csharp
#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)
```

```
 81                    {
 82                        return array;
 83                    }
 84                    else
 85                    {
 86                        return CopyFilledElements(array, filledElementsCount);
 87                    }
 88            }
 89
 90            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 91            private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
 92            {
 93                var finalArray = new TLink[filledElementsCount];
 94                for (int i = 0, j = 0; i < array.Length; i++)
 95                {
 96                    if (!_equalityComparer.Equals(array[i], default))
 97                    {
 98                        finalArray[j] = array[i];
 99                        j++;
100                    }
101            }
102 #if USEARRAYPOOL
103                ArrayPool.Free(array);
104 #endif
105                return finalArray;
106            }
107
108            [MethodImpl(MethodImplOptions.AggressiveInlining)]
109            private static int CountFilledElements(TLink[] array)
110            {
111                var count = 0;
112                for (var i = 0; i < array.Length; i++)
113                {
114                    if (!_equalityComparer.Equals(array[i], default))
115                    {
116                        count++;
117                    }
118                }
119                return count;
120            }
121        }
122 }
```

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

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

```
32                  }
33              }
34          }
35      }
36  }
```

./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs
```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Collections.Stacks;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences.Walkers
9   {
10      public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
        ↪  ISequenceWalker<TLink>
11      {
12          private readonly IStack<TLink> _stack;
13          private readonly Func<TLink, bool> _isElement;
14
15          protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links)
16          {
17              _stack = stack;
18              _isElement = isElement;
19          }
20
21          protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
        ↪  stack, links.IsPartialPoint)
22          {
23          }
24
25          public IEnumerable<TLink> Walk(TLink sequence)
26          {
27              _stack.Clear();
28              var element = sequence;
29              if (IsElement(element))
30              {
31                  yield return element;
32              }
33              else
34              {
35                  while (true)
36                  {
37                      if (IsElement(element))
38                      {
39                          if (_stack.IsEmpty)
40                          {
41                              break;
42                          }
43                          element = _stack.Pop();
44                          foreach (var output in WalkContents(element))
45                          {
46                              yield return output;
47                          }
48                          element = GetNextElementAfterPop(element);
49                      }
50                      else
51                      {
52                          _stack.Push(element);
53                          element = GetNextElementAfterPush(element);
54                      }
55                  }
56              }
57          }
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          protected abstract TLink GetNextElementAfterPop(TLink element);
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          protected abstract TLink GetNextElementAfterPush(TLink element);
67
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          protected abstract IEnumerable<TLink> WalkContents(TLink element);
```

```
  70        }
  71    }
```

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

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

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

```csharp
   1   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   2
   3   namespace Platform.Data.Doublets.Stacks
   4   {
   5       public static class StackExtensions
   6       {
   7           public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
   8           {
   9               var stackPoint = links.CreatePoint();
  10               var stack = links.Update(stackPoint, stackMarker, stackPoint);
  11               return stack;
  12           }
  13       }
  14   }
```

./Platform.Data.Doublets/SynchronizedLinks.cs

```csharp
   1   using System;
   2   using System.Collections.Generic;
   3   using Platform.Data.Doublets;
   4   using Platform.Threading.Synchronization;
   5
   6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   7
   8   namespace Platform.Data.Doublets
   9   {
  10       /// <remarks>
  11       /// TODO: Autogeneration of synchronized wrapper (decorator).
```

```csharp
        /// 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/UInt64LinksExtensions.cs

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

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

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

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

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

```csharp
                        var target = new Link<ulong>(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

```csharp
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
```

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

            public readonly ulong TransactionId;
            public readonly Link<ulong> Before;
            public readonly Link<ulong> After;
            public readonly Timestamp Timestamp;

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

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

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

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

        /// <remarks>
        /// Другие варианты реализации транзакций (атомарности):
        ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
        ↪ Target)) и индексов.
        ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
        ↪ потребуется решить вопрос
```

```
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            {
183                _layer.RevertTransition(transitionsToRevert[i]);
184            }
185            IsReverted = true;
186        }
187
188        public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
           ↪    Transaction transaction)
189        {
190            layer._currentTransactionId = layer._lastCommitedTransactionId + 1;
191            layer._currentTransactionTransitions = transaction._transitions;
192            layer._currentTransaction = transaction;
```

```csharp
            }

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

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

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

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

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

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

    public override ulong Create(IList<ulong> restrictions)
    {
```

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

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

            public override void Delete(IList<ulong> restrictions)
            {
                    var link = restrictions[Constants.IndexPart];
                    var deletedLink = new Link<ulong>(Links.GetLink(link));
                    Links.Delete(link);
                    CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                    ↪  deletedLink, default));
            }

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

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

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

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

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

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

```
344            }
345
346            private void TransitionsPusher()
347            {
348                while (!IsDisposed && _transitionsPusher != null)
349                {
350                    Thread.Sleep(DefaultPushDelay);
351                    PushTransitions();
352                }
353            }
354
355            public Transaction BeginTransaction() => new Transaction(this);
356
357            private void DisposeTransitions()
358            {
359                try
360                {
361                    var pusher = _transitionsPusher;
362                    if (pusher != null)
363                    {
364                        _transitionsPusher = null;
365                        pusher.Wait();
366                    }
367                    if (_transitions != null)
368                    {
369                        PushTransitions();
370                    }
371                    _log.DisposeIfPossible();
372                    FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
373                }
374                catch (Exception ex)
375                {
376                    ex.Ignore();
377                }
378            }
379
380            #region DisposalBase
381
382            protected override void Dispose(bool manual, bool wasDisposed)
383            {
384                if (!wasDisposed)
385                {
386                    DisposeTransitions();
387                }
388                base.Dispose(manual, wasDisposed);
389            }
390
391            #endregion
392        }
393    }
```

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

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

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

```csharp
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

```csharp
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;
        }

        public void Init()
        {
            if (_initialized)
            {
                return;
            }
            _initialized = true;
            var firstLink = _links.CreatePoint();
            if (firstLink != FirstCharLink)
            {
```

```
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 (O(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];
102         for (var i = 0; i < count; i++)
103         {
104             linksSequence[i] = FromCharToLink(chars[i]);
105         }
106         return linksSequence;
107     }
108
109     public static ulong[] FromStringToLinkArray(string sequence)
110     {
111         // char array to ulong array
112         var linksSequence = new ulong[sequence.Length];
113         for (var i = 0; i < sequence.Length; i++)
```

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

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

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

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

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

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

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

```
14        }
15    }
```

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

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

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

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

```
41                        result = comparer1.Compare(x, y) >= 0;
42                    }
43                });

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

47                var ts3 = Performance.Measure(() =>
48                {
49                    for (int i = 0; i < N; i++)
50                    {
51                        result = compareReference(x, y) >= 0;
52                    }
53                });

55                var comparer2 = new UInt64Comparer();

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

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

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

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

6   namespace Platform.Data.Doublets.Tests
7   {
8       public static class EqualityTests
9       {
10          protected class UInt64EqualityComparer : IEqualityComparer<ulong>
11          {
12              public bool Equals(ulong x, ulong y) => x == y;

14              public int GetHashCode(ulong obj) => obj.GetHashCode();
15          }

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

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

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

23          [Fact]
24          public static void EqualsPerfomanceTest()
25          {
26              const int N = 1000000;

28              ulong x = 10;
29              ulong y = 500;

31              bool result = false;

33              var ts1 = Performance.Measure(() =>
34              {
35                  for (int i = 0; i < N; i++)
36                  {
37                      result = Equals1(x, y);
38                  }
39              });

41              var ts2 = Performance.Measure(() =>
42              {
43                  for (int i = 0; i < N; i++)
44                  {
45                      result = Equals2(x, y);
46                  }
47              });

49              var ts3 = Performance.Measure(() =>
50              {
```

```
51              for (int i = 0; i < N; i++)
52              {
53                  result = Equals3(x, y);
54              }
55          });

57          var equalityComparer1 = EqualityComparer<ulong>.Default;

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

67          var equalityComparer2 = new UInt64EqualityComparer();

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

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

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

87          var comparer = Comparer<ulong>.Default;

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

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

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

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

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

8  namespace Platform.Data.Doublets.Tests
9  {
10     public unsafe static class GenericLinksTests
11     {
12         [Fact]
13         public static void CRUDTest()
14         {
15             Using<byte>(links => links.TestCRUDOperations());
16             Using<ushort>(links => links.TestCRUDOperations());
17             Using<uint>(links => links.TestCRUDOperations());
18             Using<ulong>(links => links.TestCRUDOperations());
19         }

21         [Fact]
22         public static void RawNumbersCRUDTest()
23         {
```

```
24          Using<byte>(links => links.TestRawNumbersCRUDOperations());
25          Using<ushort>(links => links.TestRawNumbersCRUDOperations());
26          Using<uint>(links => links.TestRawNumbersCRUDOperations());
27          Using<ulong>(links => links.TestRawNumbersCRUDOperations());
28      }
29
30      [Fact]
31      public static void MultipleRandomCreationsAndDeletionsTest()
32      {
33          Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
            ↪  MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
            ↪  implementation of tree cuts out 5 bits from the address space.
34          Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
            ↪  stMultipleRandomCreationsAndDeletions(100));
35          Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
            ↪  MultipleRandomCreationsAndDeletions(100));
36          Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
            ↪  tMultipleRandomCreationsAndDeletions(100));
37      }
38
39      private static void Using<TLink>(Action<ILinks<TLink>> action)
40      {
41          using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  ResizableDirectMemoryLinks<TLink>>>())
42          {
43              action(scope.Use<ILinks<TLink>>());
44          }
45      }
46  }
47 }
```

./Platform.Data.Doublets.Tests/LinksConstantsTests.cs

```
1  using Xunit;
2
3  namespace Platform.Data.Doublets.Tests
4  {
5      public static class LinksConstantsTests
6      {
7          [Fact]
8          public static void ExternalReferencesTest()
9          {
10             LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
            ↪  (long.MaxValue + 1UL, ulong.MaxValue));
11
12             //var minimum = new Hybrid<ulong>(0, isExternal: true);
13             var minimum = new Hybrid<ulong>(1, isExternal: true);
14             var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);
15
16             Assert.True(constants.IsExternalReference(minimum));
17             Assert.True(constants.IsExternalReference(maximum));
18         }
19     }
20 }
```

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

```
1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using Xunit;
5  using Platform.Data.Doublets.Sequences;
6  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8  using Platform.Data.Doublets.Sequences.Converters;
9  using Platform.Data.Doublets.PropertyOperators;
10 using Platform.Data.Doublets.Incrementers;
11 using Platform.Data.Doublets.Sequences.Walkers;
12 using Platform.Data.Doublets.Sequences.Indexes;
13 using Platform.Data.Doublets.Unicode;
14 using Platform.Data.Doublets.Numbers.Unary;
15 using Platform.Memory;
16 using Platform.Data.Doublets.ResizableDirectMemory;
17 using Platform.Data.Doublets.Decorators;
18 using Platform.Data.Doublets.ResizableDirectMemory.Specific;
19 using Platform.Data.Doublets.Numbers.Raw;
20 using Platform.Collections.Stacks;
21
22 namespace Platform.Data.Doublets.Tests
23 {
24     public static class OptimalVariantSequenceTests
```

```csharp
    {
        private static readonly string _sequenceExample = "зеленела зелёная зелень";
        private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet,
↪  consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
↪  magna aliqua.
Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
Et malesuada fames ac turpis egestas sed.
Eget velit aliquet sagittis id consectetur purus.
Dignissim cras tincidunt lobortis feugiat vivamus.
Vitae aliquet nec ullamcorper sit.
Lectus quam id leo in vitae.
Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
Integer eget aliquet nibh praesent tristique.
Vitae congue eu consequat ac felis donec et odio.
Tristique et egestas quis ipsum suspendisse.
Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
Imperdiet proin fermentum leo vel orci.
In ante metus dictum at tempor commodo.
Nisi lacus sed viverra tellus in.
Quam vulputate dignissim suspendisse in.
Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
Gravida cum sociis natoque penatibus et magnis dis parturient.
Risus quis varius quam quisque id diam.
Congue nisi vitae suscipit tellus mauris a diam maecenas.
Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
Pharetra vel turpis nunc eget lorem dolor sed viverra.
Mattis pellentesque id nibh tortor id aliquet.
Purus non enim praesent elementum facilisis leo vel.
Etiam sit amet nisl purus in mollis nunc sed.
Tortor at auctor urna nunc id cursus metus aliquam.
Volutpat odio facilisis mauris sit amet.
Turpis egestas pretium aenean pharetra magna ac placerat.
Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
Porttitor leo a diam sollicitudin tempor id eu.
Volutpat sed cras ornare arcu dui.
Ut aliquam purus sit amet luctus venenatis lectus magna.
Aliquet risus feugiat in ante metus dictum at.
Mattis nunc sed blandit libero.
Elit pellentesque habitant morbi tristique senectus et netus.
Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
Diam donec adipiscing tristique risus nec feugiat.
Pulvinar mattis nunc sed blandit libero volutpat.
Cras fermentum odio eu feugiat pretium nibh ipsum.
In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
A iaculis at erat pellentesque.
Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
Eget lorem dolor sed viverra ipsum nunc.
Leo a diam sollicitudin tempor id eu.
Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";

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

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

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

                var unaryNumberToAddressConverter = new
↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
                var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
                var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
                var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
↪  frequencyPropertyMarker, frequencyMarker);
                var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
↪  frequencyPropertyOperator, frequencyIncrementer);
```

```csharp
                    var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
                    var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
                    var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);

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

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

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

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

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

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

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

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

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

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

        private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
        ↪  SequenceToItsLocalElementLevelsConverter<ulong>
        ↪  sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
        ↪  OptimalVariantConverter<ulong> optimalVariantConverter)
        {
            index.Add(sequence);

            var optimalVariant = optimalVariantConverter.Convert(sequence);

            var readSequence1 = sequences.ToList(optimalVariant);

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

        [Fact]
        public static void SavedSequencesOptimizationTest()
        {
            LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
            ↪  (long.MaxValue + 1UL, ulong.MaxValue));

            using (var memory = new HeapResizableDirectMemory())
            using (var disposableLinks = new UInt64ResizableDirectMemoryLinks(memory,
            ↪  UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep, constants,
            ↪  useAvlBasedIndex: false))
            {
```

```
156                    var links = new UInt64Links(disposableLinks);
157
158                    var root = links.CreatePoint();
159
160                    //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
161                    var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
162
163                    var unicodeSymbolMarker = links.GetOrCreate(root,
     ↪    addressToNumberConverter.Convert(1));
164                    var unicodeSequenceMarker = links.GetOrCreate(root,
     ↪    addressToNumberConverter.Convert(2));
165
166                    var totalSequenceSymbolFrequencyCounter = new
     ↪    TotalSequenceSymbolFrequencyCounter<ulong>(links);
167                    var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
     ↪    totalSequenceSymbolFrequencyCounter);
168                    var index = new
     ↪    CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
169                    var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
     ↪    ncyNumberConverter<ulong>(linkFrequenciesCache);
170                    var sequenceToItsLocalElementLevelsConverter = new
     ↪    SequenceToItsLocalElementLevelsConverter<ulong>(links,
     ↪    linkToItsFrequencyNumberConverter);
171                    var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
     ↪    sequenceToItsLocalElementLevelsConverter);
172
173                    var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
     ↪    (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
174
175                    var unicodeSequencesOptions = new SequencesOptions<ulong>()
176                    {
177                        UseSequenceMarker = true,
178                        SequenceMarkerLink = unicodeSequenceMarker,
179                        UseIndex = true,
180                        Index = index,
181                        LinksToSequenceConverter = optimalVariantConverter,
182                        Walker = walker
183                    };
184
185                    var unicodeSequences = new Sequences.Sequences(new
     ↪    SynchronizedLinks<ulong>(links), unicodeSequencesOptions);
186
187                    // Create some sequences
188                    var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
     ↪    StringSplitOptions.RemoveEmptyEntries);
189                    var arrays = strings.Select(x => x.Select(y =>
     ↪    addressToNumberConverter.Convert(y)).ToArray()).ToArray();
190                    for (int i = 0; i < arrays.Length; i++)
191                    {
192                        unicodeSequences.Create(arrays[i].ConvertToRestrictionsValues());
193                    }
194
195                    var linksCountAfterCreation = links.Count();
196
197                    // get list of sequences links
198                    // for each sequence link
199                    //    create new sequence version
200                    //    if new sequence is not the same as sequence link
201                    //       delete sequence link
202                    //       collect garbadge
203                    //unicodeSequences.CompactAll();
204
205                    //var linksCountAfterCompactification = links.Count();
206
207                    //Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
208                }
209            }
210        }
211    }
```

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

```
1    using System;
2    using System.Collections.Generic;
3    using System.Diagnostics;
4    using System.Linq;
5    using Xunit;
6    using Platform.Data.Sequences;
7    using Platform.Data.Doublets.Sequences.Converters;
8    using Platform.Data.Doublets.Sequences.Walkers;
```

```csharp
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

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

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();
            }
```

```
21            File.Delete(tempFilename);
22        }

24        [Fact]
25        public static void BasicHeapMemoryTest()
26        {
27            using (var memory = new
              ↪  HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
28            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
              ↪  UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
29            {
30                memoryAdapter.TestBasicMemoryOperations();
31            }
32        }

34        private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
35        {
36            var link = memoryAdapter.Create();
37            memoryAdapter.Delete(link);
38        }

40        [Fact]
41        public static void NonexistentReferencesHeapMemoryTest()
42        {
43            using (var memory = new
              ↪  HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
44            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
              ↪  UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
45            {
46                memoryAdapter.TestNonexistentReferences();
47            }
48        }

50        private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
51        {
52            var link = memoryAdapter.Create();
53            memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
54            var resultLink = _constants.Null;
55            memoryAdapter.Each(foundLink =>
56            {
57                resultLink = foundLink[_constants.IndexPart];
58                return _constants.Break;
59            }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
60            Assert.True(resultLink == link);
61            Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
62            memoryAdapter.Delete(link);
63        }
64    }
65 }
```

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

```
1  using Xunit;
2  using Platform.Scopes;
3  using Platform.Memory;
4  using Platform.Data.Doublets.Decorators;
5  using Platform.Reflection;
6  using Platform.Data.Doublets.ResizableDirectMemory.Generic;
7  using Platform.Data.Doublets.ResizableDirectMemory.Specific;

9  namespace Platform.Data.Doublets.Tests
10 {
11     public static class ScopeTests
12     {
13         [Fact]
14         public static void SingleDependencyTest()
15         {
16             using (var scope = new Scope())
17             {
18                 scope.IncludeAssemblyOf<IMemory>();
19                 var instance = scope.Use<IDirectMemory>();
20                 Assert.IsType<HeapResizableDirectMemory>(instance);
21             }
22         }

24         [Fact]
25         public static void CascadeDependencyTest()
26         {
27             using (var scope = new Scope())
28             {
```

```
29              scope.Include<TemporaryFileMappedResizableDirectMemory>();
30              scope.Include<UInt64ResizableDirectMemoryLinks>();
31              var instance = scope.Use<ILinks<ulong>>();
32              Assert.IsType<UInt64ResizableDirectMemoryLinks>(instance);
33          }
34      }
35
36      [Fact]
37      public static void FullAutoResolutionTest()
38      {
39          using (var scope = new Scope(autoInclude: true, autoExplore: true))
40          {
41              var instance = scope.Use<UInt64Links>();
42              Assert.IsType<UInt64Links>(instance);
43          }
44      }
45
46      [Fact]
47      public static void TypeParametersTest()
48      {
49          using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  ResizableDirectMemoryLinks<ulong>>>())
50          {
51              var links = scope.Use<ILinks<ulong>>();
52              Assert.IsType<ResizableDirectMemoryLinks<ulong>>(links);
53          }
54      }
55  }
56 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Collections;
7  using Platform.Random;
8  using Platform.IO;
9  using Platform.Singletons;
10 using Platform.Data.Doublets.Sequences;
11 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
12 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
13 using Platform.Data.Doublets.Sequences.Converters;
14 using Platform.Data.Doublets.Unicode;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class SequencesTests
19     {
20         private static readonly LinksConstants<ulong> _constants =
            ↪  Default<LinksConstants<ulong>>.Instance;
21
22         static SequencesTests()
23         {
24             // Trigger static constructor to not mess with perfomance measurements
25             _ = BitString.GetBitMaskFromIndex(1);
26         }
27
28         [Fact]
29         public static void CreateAllVariantsTest()
30         {
31             const long sequenceLength = 8;
32
33             using (var scope = new TempLinksTestScope(useSequences: true))
34             {
35                 var links = scope.Links;
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++)
113             {
114                 sequence[i] = links.Create();
115             }
116
117             var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
118
119             //for (int i = 0; i < createResults.Length; i++)
120             //    sequences.Create(createResults[i]);
121
122             var sw0 = Stopwatch.StartNew();
123             var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
124
125             var sw1 = Stopwatch.StartNew();
126             var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
127
```

```csharp
                    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());

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

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

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

```
205
206            using (var scope = new TempLinksTestScope(useSequences: true))
207            {
208                var links = scope.Links;
209                var sequences = scope.Sequences;
210
211                var sequence = new ulong[sequenceLength];
212                for (var i = 0; i < sequenceLength; i++)
213                {
214                    sequence[i] = links.Create();
215                }
216
217                var createResults = sequences.CreateAllVariants2(sequence);
218
219                //var createResultsStrings = createResults.Select(x => x + ": " +
          ↪    sequences.FormatSequence(x)).ToList();
220                //Global.Trash = createResultsStrings;
221
222                var partialSequence = new ulong[sequenceLength - 2];
223
224                Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
225
226                var sw1 = Stopwatch.StartNew();
227                var searchResults1 =
          ↪    sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
228
229                var sw2 = Stopwatch.StartNew();
230                var searchResults2 =
          ↪    sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
231
232                //var sw3 = Stopwatch.StartNew();
233                //var searchResults3 =
          ↪    sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
234
235                var sw4 = Stopwatch.StartNew();
236                var searchResults4 =
          ↪    sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
237
238                //Global.Trash = searchResults3;
239
240                //var searchResults1Strings = searchResults1.Select(x => x + ": " +
          ↪    sequences.FormatSequence(x)).ToList();
241                //Global.Trash = searchResults1Strings;
242
243                var intersection1 = createResults.Intersect(searchResults1).ToList();
244                Assert.True(intersection1.Count == createResults.Length);
245
246                var intersection2 = createResults.Intersect(searchResults2).ToList();
247                Assert.True(intersection2.Count == createResults.Length);
248
249                var intersection4 = createResults.Intersect(searchResults4).ToList();
250                Assert.True(intersection4.Count == createResults.Length);
251
252                for (var i = 0; i < sequenceLength; i++)
253                {
254                    links.Delete(sequence[i]);
255                }
256            }
257        }
258
259        [Fact]
260        public static void BalancedPartialVariantsSearchTest()
261        {
262            const long sequenceLength = 200;
263
264            using (var scope = new TempLinksTestScope(useSequences: true))
265            {
266                var links = scope.Links;
267                var sequences = scope.Sequences;
268
269                var sequence = new ulong[sequenceLength];
270                for (var i = 0; i < sequenceLength; i++)
271                {
272                    sequence[i] = links.Create();
273                }
274
275                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
276
277                var balancedVariant = balancedVariantConverter.Convert(sequence);
278
```

```csharp
                    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()
        {
```

```csharp
            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.c
→ om/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.githubuserc
→ ontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)

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

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

...

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

        private static readonly string _exampleLoremIpsumText =

```
435                @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
      ↪    incididunt ut labore et dolore magna aliqua.
436   Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
      ↪    consequat.";
437
438        [Fact]
439        public static void CompressionTest()
440        {
441            using (var scope = new TempLinksTestScope(useSequences: true))
442            {
443                var links = scope.Links;
444                var sequences = scope.Sequences;
445
446                var e1 = links.Create();
447                var e2 = links.Create();
448
449                var sequence = new[]
450                {
451                    e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
452                };
453
454                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
455                var totalSequenceSymbolFrequencyCounter = new
                    ↪    TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
456                var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
                    ↪    totalSequenceSymbolFrequencyCounter);
457                var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
                    ↪    balancedVariantConverter, doubletFrequenciesCache);
458
459                var compressedVariant = compressingConverter.Convert(sequence);
460
461                // 1: [1]       (1->1) point
462                // 2: [2]       (2->2) point
463                // 3: [1,2]     (1->2) doublet
464                // 4: [1,2,1,2] (3->3) doublet
465
466                Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
467                Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
468                Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
469                Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
470
471                var source = _constants.SourcePart;
472                var target = _constants.TargetPart;
473
474                Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
475                Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
476                Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
477                Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
478
479                // 4 - length of sequence
480                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
                    ↪    == sequence[0]);
481                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
                    ↪    == sequence[1]);
482                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
                    ↪    == sequence[2]);
483                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
                    ↪    == sequence[3]);
484            }
485        }
486
487        [Fact]
488        public static void CompressionEfficiencyTest()
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    {
```

```
                    compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
                }

            var elapsed2 = sw2.Elapsed;

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

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

            var sw3 = Stopwatch.StartNew();

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

            var elapsed3 = sw3.Elapsed;

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

            // Assert.True(elapsed1 > elapsed2);

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

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

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

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

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

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

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

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

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

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

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

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

                    var duplicates1 = duplicateCounter1.Count();

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

                    var duplicates2 = duplicateCounter2.Count();

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

                    var duplicates3 = duplicateCounter3.Count();

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

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

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

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

                var strings = new List<string>();

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

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

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

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

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

                    var sw1 = Stopwatch.StartNew();

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

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

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

```csharp
            //    }
            //}

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

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

            var elapsed1 = sw1.Elapsed;

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

            var sw2 = Stopwatch.StartNew();

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

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

            var elapsed2 = sw2.Elapsed;

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

            Assert.True(elapsed1 > elapsed2);

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

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

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

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

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

```csharp
                }
            }

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

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

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

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

            var strings = new List<string>();

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

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

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

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

                var compressor1 = scope1.Sequences;
                var compressor2 = scope2.Sequences;

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

                var sw1 = Stopwatch.StartNew();

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

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

                var elapsed1 = sw1.Elapsed;

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

                var sw2 = Stopwatch.StartNew();

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

                var elapsed2 = sw2.Elapsed;

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

                Assert.True(elapsed1 > elapsed2);

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

                    if (sequence1 != _constants.Null && sequence2 != _constants.Null)
                    {
```

```csharp
                      var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                      ↪    scope1.Links);

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

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

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

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

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

              //compressor1.ValidateFrequencies();
          }
      }

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

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

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

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

              var createResults = sequences.CreateAllVariants2(sequence);

              Global.Trash = createResults;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

                var createResults = sequences.CreateAllVariants2(sequence);

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

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

                    sequences.CalculateAllUsages(linksTotalUsages1);

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

                    sequences.CalculateAllUsages2(linksTotalUsages2);

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

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

```csharp
1   using System.IO;
2   using Platform.Disposables;
3   using Platform.Data.Doublets.Sequences;
4   using Platform.Data.Doublets.Decorators;
5   using Platform.Data.Doublets.ResizableDirectMemory.Specific;
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  }
```

```csharp
1   using System.Collections.Generic;
2   using Xunit;
3   using Platform.Ranges;
4   using Platform.Numbers;
5   using Platform.Random;
6   using Platform.Setters;
7
8   namespace Platform.Data.Doublets.Tests
9   {
10      public static class TestExtensions
11      {
12          public static void TestCRUDOperations<T>(this ILinks<T> links)
13          {
14              var constants = links.Constants;
15
16              var equalityComparer = EqualityComparer<T>.Default;
17
18              // Create Link
19              Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
20
```

```
21          var setter = new Setter<T>(constants.Null);
22          links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
23
24          Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
25
26          var linkAddress = links.Create();
27
28          var link = new Link<T>(links.GetLink(linkAddress));
29
30          Assert.True(link.Count == 3);
31          Assert.True(equalityComparer.Equals(link.Index, linkAddress));
32          Assert.True(equalityComparer.Equals(link.Source, constants.Null));
33          Assert.True(equalityComparer.Equals(link.Target, constants.Null));
34
35          Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.One));
36
37          // Get first link
38          setter = new Setter<T>(constants.Null);
39          links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
40
41          Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
42
43          // Update link to reference itself
44          links.Update(linkAddress, linkAddress, linkAddress);
45
46          link = new Link<T>(links.GetLink(linkAddress));
47
48          Assert.True(equalityComparer.Equals(link.Source, linkAddress));
49          Assert.True(equalityComparer.Equals(link.Target, linkAddress));
50
51          // Update link to reference null (prepare for delete)
52          var updated = links.Update(linkAddress, constants.Null, constants.Null);
53
54          Assert.True(equalityComparer.Equals(updated, linkAddress));
55
56          link = new Link<T>(links.GetLink(linkAddress));
57
58          Assert.True(equalityComparer.Equals(link.Source, constants.Null));
59          Assert.True(equalityComparer.Equals(link.Target, constants.Null));
60
61          // Delete link
62          links.Delete(linkAddress);
63
64          Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
65
66          setter = new Setter<T>(constants.Null);
67          links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
68
69          Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
70      }
71
72      public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
73      {
74          // Constants
75          var constants = links.Constants;
76          var equalityComparer = EqualityComparer<T>.Default;
77
78          var h106E = new Hybrid<T>(106L, isExternal: true);
79          var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
80          var h108E = new Hybrid<T>(-108L);
81
82          Assert.Equal(106L, h106E.AbsoluteValue);
83          Assert.Equal(107L, h107E.AbsoluteValue);
84          Assert.Equal(108L, h108E.AbsoluteValue);
85
86          // Create Link (External -> External)
87          var linkAddress1 = links.Create();
88
89          links.Update(linkAddress1, h106E, h108E);
90
91          var link1 = new Link<T>(links.GetLink(linkAddress1));
92
93          Assert.True(equalityComparer.Equals(link1.Source, h106E));
94          Assert.True(equalityComparer.Equals(link1.Target, h108E));
95
96          // Create Link (Internal -> External)
97          var linkAddress2 = links.Create();
98
99          links.Update(linkAddress2, linkAddress1, h108E);
100
```

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

102
103            Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
104            Assert.True(equalityComparer.Equals(link2.Target, h108E));

105
106            // Create Link (Internal -> Internal)
107            var linkAddress3 = links.Create();

108
109            links.Update(linkAddress3, linkAddress1, linkAddress2);

110
111            var link3 = new Link<T>(links.GetLink(linkAddress3));

112
113            Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
114            Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));

115
116            // Search for created link
117            var setter1 = new Setter<T>(constants.Null);
118            links.Each(h106E, h108E, setter1.SetAndReturnFalse);

119
120            Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));

121
122            // Search for nonexistent link
123            var setter2 = new Setter<T>(constants.Null);
124            links.Each(h106E, h107E, setter2.SetAndReturnFalse);

125
126            Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));

127
128            // Update link to reference null (prepare for delete)
129            var updated = links.Update(linkAddress3, constants.Null, constants.Null);

130
131            Assert.True(equalityComparer.Equals(updated, linkAddress3));

132
133            link3 = new Link<T>(links.GetLink(linkAddress3));

134
135            Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
136            Assert.True(equalityComparer.Equals(link3.Target, constants.Null));

137
138            // Delete link
139            links.Delete(linkAddress3);

140
141            Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));

142
143            var setter3 = new Setter<T>(constants.Null);
144            links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);

145
146            Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
147        }

148
149    public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
        ↪  links, int maximumOperationsPerCycle)
150        {
151            var comparer = Comparer<TLink>.Default;
152            for (var N = 1; N < maximumOperationsPerCycle; N++)
153            {
154                var random = new System.Random(N);
155                var created = 0;
156                var deleted = 0;
157                for (var i = 0; i < N; i++)
158                {
159                    long linksCount = (Integer<TLink>)links.Count();
160                    var createPoint = random.NextBoolean();
161                    if (linksCount > 2 && createPoint)
162                    {
163                        var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
164                        TLink source = (Integer<TLink>)random.NextUInt64(linksAddressRange);
165                        TLink target = (Integer<TLink>)random.NextUInt64(linksAddressRange);
                            ↪  //-V3086
166                        var resultLink = links.CreateAndUpdate(source, target);
167                        if (comparer.Compare(resultLink, (Integer<TLink>)linksCount) > 0)
168                        {
169                            created++;
170                        }
171                    }
172                    else
173                    {
174                        links.Create();
175                        created++;
176                    }
177                }
```

```
178              Assert.True(created == (Integer<TLink>)links.Count());
179              for (var i = 0; i < N; i++)
180              {
181                  TLink link = (Integer<TLink>)(i + 1);
182                  if (links.Exists(link))
183                  {
184                      links.Delete(link);
185                      deleted++;
186                  }
187              }
188              Assert.True((Integer<TLink>)links.Count() == 0);
189          }
190      }
191  }
192 }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Diagnostics;
4   using System.IO;
5   using System.Text;
6   using System.Threading;
7   using System.Threading.Tasks;
8   using Xunit;
9   using Platform.Disposables;
10  using Platform.IO;
11  using Platform.Ranges;
12  using Platform.Random;
13  using Platform.Timestamps;
14  using Platform.Reflection;
15  using Platform.Singletons;
16  using Platform.Scopes;
17  using Platform.Counters;
18  using Platform.Diagnostics;
19  using Platform.Memory;
20  using Platform.Data.Doublets.Decorators;
21  using Platform.Data.Doublets.ResizableDirectMemory.Specific;
22
23  namespace Platform.Data.Doublets.Tests
24  {
25      public static class UInt64LinksTests
26      {
27          private static readonly LinksConstants<ulong> _constants =
            ↪  Default<LinksConstants<ulong>>.Instance;
28
29          private const long Iterations = 10 * 1024;
30
31          #region Concept
32
33          [Fact]
34          public static void MultipleCreateAndDeleteTest()
35          {
36              using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪  UInt64ResizableDirectMemoryLinks>>())
37              {
38                  new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeleti ⏎
                    ↪  ons(100);
39              }
40          }
41
42          [Fact]
43          public static void CascadeUpdateTest()
44          {
45              var itself = _constants.Itself;
46
47              using (var scope = new TempLinksTestScope(useLog: true))
48              {
49                  var links = scope.Links;
50
51                  var l1 = links.Create();
52                  var l2 = links.Create();
53
54                  l2 = links.Update(l2, l2, l1, l2);
55
56                  links.CreateAndUpdate(l2, itself);
57                  links.CreateAndUpdate(l2, itself);
58
59                  l2 = links.Update(l2, l1);
60
61                  links.Delete(l2);
```

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

                links.Unsync.DisposeIfPossible(); // Close links to access log

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

        [Fact]
        public static void BasicTransactionLogTest()
        {
            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;
                var l1 = links.Create();
                var l2 = links.Create();

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

                links.Delete(l1);

                links.Unsync.DisposeIfPossible(); // Close links to access log

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

        [Fact]
        public static void TransactionAutoRevertedTest()
        {
            // Auto Reverted (Because no commit at transaction)
            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;
                var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
                using (var transaction = transactionsLayer.BeginTransaction())
                {
                    var l1 = links.Create();
                    var l2 = links.Create();

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

                Assert.Equal(0UL, links.Count());

                links.Unsync.DisposeIfPossible();

                var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
                ↪   cope.TempTransactionLogFilename);
                Assert.Single(transitions);
            }
        }

        [Fact]
        public static void TransactionUserCodeErrorNoDataSavedTest()
        {
            // User Code Error (Autoreverted), no data saved
            var itself = _constants.Itself;

            TempLinksTestScope lastScope = null;
            try
            {
                using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                ↪   useLog: true))
                {
                    var links = scope.Links;
                    var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
                    ↪   atorBase<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);
```

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

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

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

142                     links.Delete(l2);

144                     ExceptionThrower();

146                     transaction.Commit();
147                 }

149                 Global.Trash = links.Count();
150             }
151         }
152         catch
153         {
154             Assert.False(lastScope == null);

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

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

160             lastScope.DeleteFiles();
161         }
162     }

164     [Fact]
165     public static void TransactionUserCodeErrorSomeDataSavedTest()
166     {
167         // User Code Error (Autoreverted), some data saved
168         var itself = _constants.Itself;

170         TempLinksTestScope lastScope = null;
171         try
172         {
173             ulong l1;
174             ulong l2;

176             using (var scope = new TempLinksTestScope(useLog: true))
177             {
178                 var links = scope.Links;
179                 l1 = links.CreateAndUpdate(itself, itself);
180                 l2 = links.CreateAndUpdate(itself, itself);

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

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

187                 links.Unsync.DisposeIfPossible();

189                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
                    ↪    scope.TempTransactionLogFilename);
190             }

192             using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                ↪    useLog: true))
193             {
194                 var links = scope.Links;
195                 var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
196                 using (var transaction = transactionsLayer.BeginTransaction())
197                 {
198                     l2 = links.Update(l2, l1);

200                     links.Delete(l2);

202                     ExceptionThrower();

204                     transaction.Commit();
205                 }

207                 Global.Trash = links.Count();
208             }
209         }
210         catch
```

```csharp
    {
        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();
        }

        Global.Trash = links.Count();
    }

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

    // Damage database

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

```csharp
                    // Try load damaged database
                    try
                    {
                        // TODO: Fix
                        using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                        ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                        ↪  tempTransactionLogFilename))
                        using (var links = new UInt64Links(memoryAdapter))
                        {
                            Global.Trash = links.Count();
                        }
                    }
                    catch (NotSupportedException ex)
                    {
                        Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
                        ↪  yet.");
                    }

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

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

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

                    var itself = _constants.Itself;

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

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

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

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

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

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

                                links.Delete(l2);

                                ExceptionThrower();

                                transaction.Commit();
                            }

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

```csharp
351             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp↵
      ↪    TransactionLogFilename);
352         }
353
354         File.Delete(tempDatabaseFilename);
355         File.Delete(tempTransactionLogFilename);
356     }
357
358     private static void ExceptionThrower() => throw new InvalidOperationException();
359
360     [Fact]
361     public static void PathsTest()
362     {
363         var source = _constants.SourcePart;
364         var target = _constants.TargetPart;
365
366         using (var scope = new TempLinksTestScope())
367         {
368             var links = scope.Links;
369             var l1 = links.CreatePoint();
370             var l2 = links.CreatePoint();
371
372             var r1 = links.GetByKeys(l1, source, target, source);
373             var r2 = links.CheckPathExistance(l2, l2, l2, l2);
374         }
375     }
376
377     [Fact]
378     public static void RecursiveStringFormattingTest()
379     {
380         using (var scope = new TempLinksTestScope(useSequences: true))
381         {
382             var links = scope.Links;
383             var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
384
385             var a = links.CreatePoint();
386             var b = links.CreatePoint();
387             var c = links.CreatePoint();
388
389             var ab = links.CreateAndUpdate(a, b);
390             var cb = links.CreateAndUpdate(c, b);
391             var ac = links.CreateAndUpdate(a, c);
392
393             a = links.Update(a, c, b);
394             b = links.Update(b, a, c);
395             c = links.Update(c, a, b);
396
397             Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
398             Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
399             Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
400
401             Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
      ↪    "(5:(4:5 (6:5 4)) 6)");
402             Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
      ↪    "(6:(5:(4:5 6) 6) 4)");
403             Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
      ↪    "(4:(5:4 (6:5 4)) 6)");
404
405             // 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)"
406
407             Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
      ↪    "{{5}{5}{4}{6}}");
408             Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
      ↪    "{{5}{6}{6}{4}}");
409             Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
      ↪    "{{4}{5}{4}{6}}");
410         }
411     }
412
413     private static void DefaultFormatter(StringBuilder sb, ulong link)
414     {
415         sb.Append(link.ToString());
416     }
417
418     #endregion
419
420     #region Performance
421
```

```
422              /*
423          public static void RunAllPerformanceTests()
424          {
425              try
426              {
427                  links.TestLinksInSteps();
428              }
429              catch (Exception ex)
430              {
431                  ex.WriteToConsole();
432              }
433
434              return;
435
436              try
437              {
438                  //ThreadPool.SetMaxThreads(2, 2);
439
440                  // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
↪   результат
441                  // Также это дополнительно помогает в отладке
442                  // Увеличивает вероятность попадания информации в кэши
443                  for (var i = 0; i < 10; i++)
444                  {
445                      //0 - 10 ГБ
446                      //Каждые 100 МБ срез цифр
447
448                      //links.TestGetSourceFunction();
449                      //links.TestGetSourceFunctionInParallel();
450                      //links.TestGetTargetFunction();
451                      //links.TestGetTargetFunctionInParallel();
452                      links.Create64BillionLinks();
453
454                      links.TestRandomSearchFixed();
455                      //links.Create64BillionLinksInParallel();
456                      links.TestEachFunction();
457                      //links.TestForeach();
458                      //links.TestParallelForeach();
459                  }
460
461                  links.TestDeletionOfAllLinks();
462
463              }
464              catch (Exception ex)
465              {
466                  ex.WriteToConsole();
467              }
468          }*/
469
470              /*
471          public static void TestLinksInSteps()
472          {
473              const long gibibyte = 1024 * 1024 * 1024;
474              const long mebibyte = 1024 * 1024;
475
476              var totalLinksToCreate = gibibyte /
↪   Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
477              var linksStep = 102 * mebibyte /
↪   Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478
479              var creationMeasurements = new List<TimeSpan>();
480              var searchMeasuremets = new List<TimeSpan>();
481              var deletionMeasurements = new List<TimeSpan>();
482
483              GetBaseRandomLoopOverhead(linksStep);
484              GetBaseRandomLoopOverhead(linksStep);
485
486              var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
487
488              ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
489
490              var loops = totalLinksToCreate / linksStep;
491
492              for (int i = 0; i < loops; i++)
493              {
494                  creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
495                  searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));
496
497                  Console.Write("\rC + S {0}/{1}", i + 1, loops);
```

```
498             }
499
500             ConsoleHelpers.Debug();
501
502             for (int i = 0; i < loops; i++)
503             {
504                 deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
505
506                 Console.Write("\rD {0}/{1}", i + 1, loops);
507             }
508
509             ConsoleHelpers.Debug();
510
511             ConsoleHelpers.Debug("C S D");
512
513             for (int i = 0; i < loops; i++)
514             {
515                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
   ↪  searchMeasuremets[i], deletionMeasurements[i]);
516             }
517
518             ConsoleHelpers.Debug("C S D (no overhead)");
519
520             for (int i = 0; i < loops; i++)
521             {
522                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
   ↪  searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
523             }
524
525             ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
   ↪  links.Total);
526         }
527
528         private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
   ↪  amountToCreate)
529         {
530             for (long i = 0; i < amountToCreate; i++)
531                 links.Create(0, 0);
532         }
533
534          private static TimeSpan GetBaseRandomLoopOverhead(long loops)
535          {
536              return Measure(() =>
537              {
538                  ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
539                  ulong result = 0;
540                  for (long i = 0; i < loops; i++)
541                  {
542                      var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
543                      var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544
545                      result += maxValue + source + target;
546                  }
547                  Global.Trash = result;
548              });
549         }
550          */
551
552         [Fact(Skip = "performance test")]
553         public static void GetSourceTest()
554         {
555             using (var scope = new TempLinksTestScope())
556             {
557                 var links = scope.Links;
558                 ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
   ↪  Iterations);
559
560                 ulong counter = 0;
561
562                 //var firstLink = links.First();
563                 // Создаём одну связь, из которой будет производить считывание
564                 var firstLink = links.Create();
565
566                 var sw = Stopwatch.StartNew();
567
568                 // Тестируем саму функцию
569                 for (ulong i = 0; i < Iterations; i++)
570                 {
571                     counter += links.GetSource(firstLink);
```

```csharp
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

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

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

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

                long counter = 0;

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

                var sw = Stopwatch.StartNew();

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

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

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

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

                ulong counter = 0;

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

                var sw = Stopwatch.StartNew();

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

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

                ConsoleHelpers.Debug(
```

```
648                "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                   ↪ second), counter result: {3}",
649                Iterations, elapsedTime, (long)iterationsPerSecond, counter);
650        }
651    }
652
653    [Fact(Skip = "performance test")]
654    public static void TestGetTargetInParallel()
655    {
656        using (var scope = new TempLinksTestScope())
657        {
658            var links = scope.Links;
659            ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
                   ↪ parallel.", Iterations);
660
661            long counter = 0;
662
663            //var firstLink = links.First();
664            var firstLink = links.Create();
665
666            var sw = Stopwatch.StartNew();
667
668            Parallel.For(0, Iterations, x =>
669            {
670                Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
671                //Interlocked.Increment(ref counter);
672            });
673
674            var elapsedTime = sw.Elapsed;
675
676            var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
677
678            links.Delete(firstLink);
679
680            ConsoleHelpers.Debug(
681                "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                   ↪ second), counter result: {3}",
682                Iterations, elapsedTime, (long)iterationsPerSecond, counter);
683        }
684    }
685
686    // TODO: Заполнить базу данных перед тестом
687    /*
688    [Fact]
689    public void TestRandomSearchFixed()
690    {
691        var tempFilename = Path.GetTempFileName();
692
693        using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
       ↪ DefaultLinksSizeStep))
694        {
695            long iterations = 64 * 1024 * 1024 /
       ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
696
697            ulong counter = 0;
698            var maxLink = links.Total;
699
700            ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
701
702            var sw = Stopwatch.StartNew();
703
704            for (var i = iterations; i > 0; i--)
705            {
706                var source =
       ↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
707                var target =
       ↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
708
709                counter += links.Search(source, target);
710            }
711
712            var elapsedTime = sw.Elapsed;
713
714            var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
715
716            ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
       ↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
       ↪ counter);
717        }
```

```csharp
                File.Delete(tempFilename);
            }*/

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

                var maxLink = links.Count();

                var iterations = links.Count();

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

                var sw = Stopwatch.StartNew();

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

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

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

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;

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

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

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

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

                var sw = Stopwatch.StartNew();

                links.Each(counter.IncrementAndReturnTrue);

                var elapsedTime = sw.Elapsed;

                var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;

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

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

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

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

```
793                    var sw = Stopwatch.StartNew();
794
795                    //foreach (var link in links)
796                    //{
797                    //    counter++;
798                    //}
799
800                    var elapsedTime = sw.Elapsed;
801
802                    var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
803
804                    ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
      links per second)", counter, elapsedTime, (long)linksPerSecond);
805                }
806
807                File.Delete(tempFilename);
808            }
809        */
810
811        /*
812        [Fact]
813        public static void TestParallelForeach()
814        {
815            var tempFilename = Path.GetTempFileName();
816
817            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
      DefaultLinksSizeStep))
818            {
819
820                long counter = 0;
821
822                ConsoleHelpers.Debug("Testing parallel foreach through links.");
823
824                var sw = Stopwatch.StartNew();
825
826                //Parallel.ForEach((IEnumerable<ulong>)links, x =>
827                //{
828                //    Interlocked.Increment(ref counter);
829                //});
830
831                var elapsedTime = sw.Elapsed;
832
833                var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
834
835                ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
      {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
836            }
837
838            File.Delete(tempFilename);
839        }
840        */
841
842        [Fact(Skip = "performance test")]
843        public static void Create64BillionLinks()
844        {
845            using (var scope = new TempLinksTestScope())
846            {
847                var links = scope.Links;
848                var linksBeforeTest = links.Count();
849
850                long linksToCreate = 64 * 1024 * 1024 /
                    UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
851
852                ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
853
854                var elapsedTime = Performance.Measure(() =>
855                {
856                    for (long i = 0; i < linksToCreate; i++)
857                    {
858                        links.Create();
859                    }
860                });
861
862                var linksCreated = links.Count() - linksBeforeTest;
863                var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
864
865                ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
866
867                ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    linksCreated, elapsedTime,
```

```
868                          (long)linksPerSecond);
869              }
870          }
871
872          [Fact(Skip = "performance test")]
873          public static void Create64BillionLinksInParallel()
874          {
875              using (var scope = new TempLinksTestScope())
876              {
877                  var links = scope.Links;
878                  var linksBeforeTest = links.Count();
879
880                  var sw = Stopwatch.StartNew();
881
882                  long linksToCreate = 64 * 1024 * 1024 /
                     ↪  UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
883
884                  ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
885
886                  Parallel.For(0, linksToCreate, x => links.Create());
887
888                  var elapsedTime = sw.Elapsed;
889
890                  var linksCreated = links.Count() - linksBeforeTest;
891                  var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
892
893                  ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                     ↪  linksCreated, elapsedTime,
894                      (long)linksPerSecond);
895              }
896          }
897
898          [Fact(Skip = "useless: O(0), was dependent on creation tests")]
899          public static void TestDeletionOfAllLinks()
900          {
901              using (var scope = new TempLinksTestScope())
902              {
903                  var links = scope.Links;
904                  var linksBeforeTest = links.Count();
905
906                  ConsoleHelpers.Debug("Deleting all links");
907
908                  var elapsedTime = Performance.Measure(links.DeleteAll);
909
910                  var linksDeleted = linksBeforeTest - links.Count();
911                  var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
912
913                  ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
                     ↪  linksDeleted, elapsedTime,
914                      (long)linksPerSecond);
915              }
916          }
917
918          #endregion
919      }
920  }
```

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

```
1   using Xunit;
2   using Platform.Random;
3   using Platform.Data.Doublets.Numbers.Unary;
4
5   namespace Platform.Data.Doublets.Tests
6   {
7       public static class UnaryNumberConvertersTests
8       {
9           [Fact]
10          public static void ConvertersTest()
11          {
12              using (var scope = new TempLinksTestScope())
13              {
14                  const int N = 10;
15                  var links = scope.Links;
16                  var meaningRoot = links.CreatePoint();
17                  var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
18                  var powerOf2ToUnaryNumberConverter = new
                     ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
19                  var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
                     ↪  powerOf2ToUnaryNumberConverter);
20                  var random = new System.Random(0);
```

```
21            ulong[] numbers = new ulong[N];
22            ulong[] unaryNumbers = new ulong[N];
23            for (int i = 0; i < N; i++)
24            {
25                numbers[i] = random.NextUInt64();
26                unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
27            }
28            var fromUnaryNumberConverterUsingOrOperation = new
   ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
   ↪  powerOf2ToUnaryNumberConverter);
29            var fromUnaryNumberConverterUsingAddOperation = new
   ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
30            for (int i = 0; i < N; i++)
31            {
32                Assert.Equal(numbers[i],
   ↪  fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
33                Assert.Equal(numbers[i],
   ↪  fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
34            }
35        }
36    }
37  }
38 }
```

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

```csharp
1  using Xunit;
2  using Platform.Interfaces;
3  using Platform.Memory;
4  using Platform.Reflection;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.Incrementers;
7  using Platform.Data.Doublets.Numbers.Raw;
8  using Platform.Data.Doublets.Numbers.Unary;
9  using Platform.Data.Doublets.PropertyOperators;
10 using Platform.Data.Doublets.Sequences.Converters;
11 using Platform.Data.Doublets.Sequences.Indexes;
12 using Platform.Data.Doublets.Sequences.Walkers;
13 using Platform.Data.Doublets.Unicode;
14 using Platform.Data.Doublets.ResizableDirectMemory.Generic;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class UnicodeConvertersTests
19     {
20         [Fact]
21         public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
22         {
23             using (var scope = new TempLinksTestScope())
24             {
25                 var links = scope.Links;
26                 var meaningRoot = links.CreatePoint();
27                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
28                 var powerOf2ToUnaryNumberConverter = new
   ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
29                 var addressToUnaryNumberConverter = new
   ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
30                 var unaryNumberToAddressConverter = new
   ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
   ↪  powerOf2ToUnaryNumberConverter);
31                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
   ↪  addressToUnaryNumberConverter, unaryNumberToAddressConverter);
32             }
33         }
34
35         [Fact]
36         public static void CharAndRawNumberUnicodeSymbolConvertersTest()
37         {
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
```

```csharp
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
```

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

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

            var resultingString =
            ↪    unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);

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

Index