# LinksPlatform's Platform.Data.Doublets Class Library

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

```csharp
1   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3   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                  {
```

```
27                    disposableDecorator.Facade = value;
28                }
29            }
30        }
31
32        [MethodImpl(MethodImplOptions.AggressiveInlining)]
33        protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
34        {
35            Constants = links.Constants;
36            Facade = this;
37        }
38
39        [MethodImpl(MethodImplOptions.AggressiveInlining)]
40        public virtual TLink Count(IList<TLink> restrictions) => Links.Count(restrictions);
41
42        [MethodImpl(MethodImplOptions.AggressiveInlining)]
43        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
   ↪   => Links.Each(handler, restrictions);
44
45        [MethodImpl(MethodImplOptions.AggressiveInlining)]
46        public virtual TLink Create(IList<TLink> restrictions) => Links.Create(restrictions);
47
48        [MethodImpl(MethodImplOptions.AggressiveInlining)]
49        public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
   ↪   Links.Update(restrictions, substitution);
50
51        [MethodImpl(MethodImplOptions.AggressiveInlining)]
52        public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);
53    }
54 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Decorators
9  {
10     public abstract class LinksDisposableDecoratorBase<TLink> : DisposableBase, ILinks<TLink>
11     {
12         private ILinks<TLink> _facade;
13
14         public LinksConstants<TLink> Constants { get; }
15
16         public ILinks<TLink> Links { get; }
17
18         public ILinks<TLink> Facade
19         {
20             get => _facade;
21             set
22             {
23                 _facade = value;
24                 if (Links is LinksDecoratorBase<TLink> decorator)
25                 {
26                     decorator.Facade = value;
27                 }
28                 else if (Links is LinksDisposableDecoratorBase<TLink> disposableDecorator)
29                 {
30                     disposableDecorator.Facade = value;
31                 }
32             }
33         }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected LinksDisposableDecoratorBase(ILinks<TLink> links)
37         {
38             Links = links;
39             Constants = links.Constants;
40             Facade = this;
41         }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public virtual TLink Count(IList<TLink> restrictions) => Links.Count(restrictions);
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         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
```

```
13        [MethodImpl(MethodImplOptions.AggressiveInlining)]
14        public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
15
16        [MethodImpl(MethodImplOptions.AggressiveInlining)]
17        public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
18        {
19            var constants = Constants;
20            var itselfConstant = constants.Itself;
21            var indexPartConstant = constants.IndexPart;
22            var sourcePartConstant = constants.SourcePart;
23            var targetPartConstant = constants.TargetPart;
24            var restrictionsCount = restrictions.Count;
25            if (!_equalityComparer.Equals(constants.Any, itselfConstant)
26             && (((restrictionsCount > indexPartConstant) &&
                ↪  _equalityComparer.Equals(restrictions[indexPartConstant], itselfConstant))
27             || ((restrictionsCount > sourcePartConstant) &&
                ↪  _equalityComparer.Equals(restrictions[sourcePartConstant], itselfConstant))
28             || ((restrictionsCount > targetPartConstant) &&
                ↪  _equalityComparer.Equals(restrictions[targetPartConstant], itselfConstant))))
29            {
30                // Itself constant is not supported for Each method right now, skipping execution
31                return constants.Continue;
32            }
33            return Links.Each(handler, restrictions);
34        }
35
36        [MethodImpl(MethodImplOptions.AggressiveInlining)]
37        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
            ↪  Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Itself,
            ↪  restrictions, substitution));
38    }
39 }
```

./Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.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      /// <remarks>
9      /// Not practical if newSource and newTarget are too big.
10     /// To be able to use practical version we should allow to create link at any specific
           ↪  location inside ResizableDirectMemoryLinks.
11     /// This in turn will require to implement not a list of empty links, but a list of ranges
           ↪  to store it more efficiently.
12     /// </remarks>
13     public class LinksNonExistentDependenciesCreator<TLink> : LinksDecoratorBase<TLink>
14     {
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public LinksNonExistentDependenciesCreator(ILinks<TLink> links) : base(links) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
20         {
21             var constants = Constants;
22             Links.EnsureCreated(substitution[constants.SourcePart],
                   ↪  substitution[constants.TargetPart]);
23             return Links.Update(restrictions, substitution);
24         }
25     }
26 }
```

./Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.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 LinksNullConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksNullConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         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
22     ///     поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
23     ///     460 752 303 423 488
24     /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
25     ///     (битовыми строками) - вариант матрицы (выстраеваемой лениво).
```

```csharp
        ///
        /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
        ↪   выбрасываться только при #if DEBUG
        /// </remarks>
        public class UInt64Links : LinksDisposableDecoratorBase<ulong>
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public UInt64Links(ILinks<ulong> links) : base(links) { }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public override ulong Create(IList<ulong> restrictions) => Links.CreatePoint();

            public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
            {
                var constants = Constants;
                var indexPartConstant = constants.IndexPart;
                var updatedLink = restrictions[indexPartConstant];
                var sourcePartConstant = constants.SourcePart;
                var newSource = substitution[sourcePartConstant];
                var targetPartConstant = constants.TargetPart;
                var newTarget = substitution[targetPartConstant];
                var nullConstant = constants.Null;
                var existedLink = nullConstant;
                var itselfConstant = constants.Itself;
                if (newSource != itselfConstant && newTarget != itselfConstant)
                {
                    existedLink = Links.SearchOrDefault(newSource, newTarget);
                }
                if (existedLink == nullConstant)
                {
                    var before = Links.GetLink(updatedLink);
                    if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
                    ↪   newTarget)
                    {
                        Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
                        ↪   newSource,
                                                 newTarget == itselfConstant ? updatedLink :
                                                 ↪   newTarget);
                    }
                    return updatedLink;
                }
                else
                {
                    return Facade.MergeAndDelete(updatedLink, existedLink);
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public override void Delete(IList<ulong> restrictions)
            {
                var linkIndex = restrictions[Constants.IndexPart];
                Links.EnforceResetValues(linkIndex);
                Facade.DeleteAllUsages(linkIndex);
                Links.Delete(linkIndex);
            }
        }
    }
```

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

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

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

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

```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
231                        if (matchHandler != null)
232                        {
233                            return substitutionHandler(before, after);
234                        }
235                        return Constants.Continue;
236                    }
237                else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
238                    {
239                        if (patternOrCondition.Count == 1)
240                        {
241                            var linkToDelete = patternOrCondition[0];
242                            var before = Links.GetLink(linkToDelete);
243                            if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                                ↪  Constants.Break))
244                            {
245                                return Constants.Break;
246                            }
247                            var after = ArrayPool<TLink>.Empty;
248                            Links.Update(linkToDelete, Constants.Null, Constants.Null);
249                            Links.Delete(linkToDelete);
250                            if (matchHandler != null)
251                            {
252                                return substitutionHandler(before, after);
253                            }
254                            return Constants.Continue;
255                        }
256                        else
257                        {
258                            throw new NotSupportedException();
259                        }
260                    }
261                else // Replace / Update
262                    {
263                        if (patternOrCondition.Count == 1) //-V3125
264                        {
265                            var linkToUpdate = patternOrCondition[0];
266                            var before = Links.GetLink(linkToUpdate);
267                            if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                                ↪  Constants.Break))
268                            {
269                                return Constants.Break;
270                            }
271                            var after = (IList<TLink>)substitution.ToArray(); //-V3125
272                            if (_equalityComparer.Equals(after[0], default))
273                            {
274                                after[0] = linkToUpdate;
275                            }
276                            if (substitution.Count == 1)
277                            {
278                                if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
279                                {
280                                    after = Links.GetLink(substitution[0]);
281                                    Links.Update(linkToUpdate, Constants.Null, Constants.Null);
282                                    Links.Delete(linkToUpdate);
283                                }
284                            }
285                            else if (substitution.Count == 3)
286                            {
287                                //Links.Update(after);
288                            }
289                            else
290                            {
291                                throw new NotSupportedException();
292                            }
293                            if (matchHandler != null)
294                            {
295                                return substitutionHandler(before, after);
296                            }
297                            return Constants.Continue;
298                        }
299                        else
300                        {
301                            throw new NotSupportedException();
302                        }
303                    }
304            }
305
306            /// <remarks>
```

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

```csharp
1   using System;
2   using System.Reflection;
3   using System.Reflection.Emit;
4   using Platform.Reflection;
5   using Platform.Converters;
6   using Platform.Exceptions;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets
11  {
12      public class Hybrid<T>
13      {
14          private static readonly Func<object, T> _absAndConvert;
15          private static readonly Func<object, T> _absAndNegateAndConvert;
16
17          static Hybrid()
18          {
19              _absAndConvert = DelegateHelpers.Compile<Func<object, T>>(emiter =>
20              {
21                  Ensure.Always.IsUnsignedInteger<T>();
22                  emiter.LoadArgument(0);
23                  var signedVersion = NumericType<T>.SignedVersion;
24                  var signedVersionField =
                      typeof(NumericType<T>).GetTypeInfo().GetField("SignedVersion",
                      BindingFlags.Static | BindingFlags.Public);
25                  //emiter.LoadField(signedVersionField);
26                  emiter.Emit(OpCodes.Ldsfld, signedVersionField);
27                  var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
                      Types<object, Type>.Array);
28                  emiter.Call(changeTypeMethod);
29                  emiter.UnboxValue(signedVersion);
30                  var absMethod = typeof(Math).GetTypeInfo().GetMethod("Abs", new[] {
                      signedVersion });
31                  emiter.Call(absMethod);
32                  var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                      signedVersion });
33                  emiter.Call(unsignedMethod);
34                  emiter.Return();
35              });
36              _absAndNegateAndConvert = DelegateHelpers.Compile<Func<object, T>>(emiter =>
37              {
38                  Ensure.Always.IsUnsignedInteger<T>();
39                  emiter.LoadArgument(0);
40                  var signedVersion = NumericType<T>.SignedVersion;
41                  var signedVersionField =
                      typeof(NumericType<T>).GetTypeInfo().GetField("SignedVersion",
                      BindingFlags.Static | BindingFlags.Public);
42                  //emiter.LoadField(signedVersionField);
43                  emiter.Emit(OpCodes.Ldsfld, signedVersionField);
44                  var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
                      Types<object, Type>.Array);
45                  emiter.Call(changeTypeMethod);
46                  emiter.UnboxValue(signedVersion);
47                  var absMethod = typeof(Math).GetTypeInfo().GetMethod("Abs", new[] {
                      signedVersion });
48                  emiter.Call(absMethod);
49                  var negateMethod = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate
                      ").MakeGenericMethod(signedVersion);
50                  emiter.Call(negateMethod);
51                  var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                      signedVersion });
52                  emiter.Call(unsignedMethod);
53                  emiter.Return();
54              });
55          }
56
57          public readonly T Value;
58          public bool IsNothing => Convert.ToInt64(To.Signed(Value)) == 0;
59          public bool IsInternal => Convert.ToInt64(To.Signed(Value)) > 0;
60          public bool IsExternal => Convert.ToInt64(To.Signed(Value)) < 0;
61          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
132                        return false;
133                    }
134                    current = next;
135                }
136                return true;
137            }
138
139            /// <remarks>
140            /// Может потребовать дополнительного стека для PathElement's при использовании
            ↪  SequenceWalker.
141            /// </remarks>
142            public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
            ↪  path)
143            {
144                links.EnsureLinkExists(root, "root");
145                var currentLink = root;
146                for (var i = 0; i < path.Length; i++)
147                {
148                    currentLink = links.GetLink(currentLink)[path[i]];
149                }
150                return currentLink;
151            }
152
153            public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
            ↪  links, TLink root, ulong size, ulong index)
154            {
155                var constants = links.Constants;
156                var source = constants.SourcePart;
157                var target = constants.TargetPart;
158                if (!Platform.Numbers.Math.IsPowerOfTwo(size))
159                {
160                    throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
                    ↪  than powers of two are not supported.");
161                }
162                var path = new BitArray(BitConverter.GetBytes(index));
163                var length = Bit.GetLowestPosition(size);
164                links.EnsureLinkExists(root, "root");
165                var currentLink = root;
166                for (var i = length - 1; i >= 0; i--)
167                {
168                    currentLink = links.GetLink(currentLink)[path[i] ? target : source];
169                }
170                return currentLink;
171            }
172
173            #endregion
174
175            /// <summary>
176            /// Возвращает индекс указанной связи.
177            /// </summary>
178            /// <param name="links">Хранилище связей.</param>
179            /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪  содержимого.</param>
180            /// <returns>Индекс начальной связи для указанной связи.</returns>
181            [MethodImpl(MethodImplOptions.AggressiveInlining)]
182            public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪  link[links.Constants.IndexPart];
183
184            /// <summary>
185            /// Возвращает индекс начальной (Source) связи для указанной связи.
186            /// </summary>
187            /// <param name="links">Хранилище связей.</param>
188            /// <param name="link">Индекс связи.</param>
189            /// <returns>Индекс начальной связи для указанной связи.</returns>
190            [MethodImpl(MethodImplOptions.AggressiveInlining)]
191            public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪  links.GetLink(link)[links.Constants.SourcePart];
192
193            /// <summary>
194            /// Возвращает индекс начальной (Source) связи для указанной связи.
195            /// </summary>
196            /// <param name="links">Хранилище связей.</param>
197            /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪  содержимого.</param>
198            /// <returns>Индекс начальной связи для указанной связи.</returns>
199            [MethodImpl(MethodImplOptions.AggressiveInlining)]
200            public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪  link[links.Constants.SourcePart];
```

```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
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
        ↪   Func<IList<TLink>, TLink> handler)
        {
            var constants = links.Constants;
            return links.Each(handler, constants.Any, source, target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
        ↪   restrictions)
        {
            long arraySize = (Integer<TLink>)links.Count(restrictions);
            var array = new IList<TLink>[arraySize];
            if (arraySize > 0)
            {
                var filler = new ArrayFiller<IList<TLink>, TLink>(array,
                ↪   links.Constants.Continue);
                links.Each(filler.AddAndReturnConstant, restrictions);
            }
            return array;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
        ↪   restrictions)
        {
            long arraySize = (Integer<TLink>)links.Count(restrictions);
            var array = new TLink[arraySize];
            if (arraySize > 0)
            {
                var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
                links.Each(filler.AddFirstAndReturnConstant, restrictions);
            }
            return array;
        }

        /// <summary>
        /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
        ↪   в хранилище связей.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Начало связи.</param>
        /// <param name="target">Конец связи.</param>
        /// <returns>Значение, определяющее существует ли связь.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;

        #region Ensure
        // TODO: May be move to EnsureExtensions or make it both there and here

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
        ↪   reference, string argumentName)
        {
            if (links.Constants.IsInnerReference(reference) && !links.Exists(reference))
            {
                throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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)
        {
            for (int i = 0; i < restrictions.Count; i++)
```

```
322             {
323                 links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
324             }
325         }
326
327         [MethodImpl(MethodImplOptions.AggressiveInlining)]
328         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
      ↪  string argumentName)
329         {
330             var equalityComparer = EqualityComparer<TLink>.Default;
331             if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
332             {
333                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
334             }
335         }
336
337         [MethodImpl(MethodImplOptions.AggressiveInlining)]
338         public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
      ↪  link, string argumentName)
339         {
340             var equalityComparer = EqualityComparer<TLink>.Default;
341             if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
342             {
343                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
344             }
345         }
346
347         /// <param name="links">Хранилище связей.</param>
348         [MethodImpl(MethodImplOptions.AggressiveInlining)]
349         public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
      ↪  TLink target)
350         {
351             if (links.Exists(source, target))
352             {
353                 throw new LinkWithSameValueAlreadyExistsException();
354             }
355         }
356
357         /// <param name="links">Хранилище связей.</param>
358         public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
359         {
360             if (links.HasUsages(link))
361             {
362                 throw new ArgumentLinkHasDependenciesException<TLink>(link);
363             }
364         }
365
366         /// <param name="links">Хранилище связей.</param>
367         public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
      ↪  addresses) => links.EnsureCreated(links.Create, addresses);
368
369         /// <param name="links">Хранилище связей.</param>
370         public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
      ↪  addresses) => links.EnsureCreated(links.CreatePoint, addresses);
371
372         /// <param name="links">Хранилище связей.</param>
373         public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
      ↪  params TLink[] addresses)
374         {
375             var constants = links.Constants;
376
377             var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
      ↪  !links.Exists(x)));
378             if (nonExistentAddresses.Count > 0)
379             {
380                 var max = nonExistentAddresses.Max();
381                 max = (Integer<TLink>)System.Math.Min((ulong)(Integer<TLink>)max,
      ↪  (ulong)(Integer<TLink>)constants.PossibleInnerReferencesRange.Maximum);
382                 var createdLinks = new List<TLink>();
383                 var equalityComparer = EqualityComparer<TLink>.Default;
384                 TLink createdLink = creator();
385                 while (!equalityComparer.Equals(createdLink, max))
386                 {
387                     createdLinks.Add(createdLink);
388                 }
389                 for (var i = 0; i < createdLinks.Count; i++)
390                 {
391                     if (!nonExistentAddresses.Contains(createdLinks[i]))
```

```csharp
                    {
                        links.Delete(createdLinks[i]);
                    }
                }
            }
        }

        #endregion

        /// <param name="links">Хранилище связей.</param>
        public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
        {
            var constants = links.Constants;
            var values = links.GetLink(link);
            TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
            ↪  constants.Any));
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (equalityComparer.Equals(values[constants.SourcePart], link))
            {
                usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
            }
            TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
            ↪  link));
            if (equalityComparer.Equals(values[constants.TargetPart], link))
            {
                usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
            }
            return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
        }

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

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

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

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

```csharp
        /// <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];
            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>
```

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

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

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

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

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

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

```
665                     {
666                         var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
667                         if (totalUsages > 0)
668                         {
669                             var usages = ArrayPool.Allocate<TLink>(totalUsages);
670                             var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
                              ↪ links.Constants.Continue);
671                             var i = 0L;
672                             if (usagesAsSourceCount > 0)
673                             {
674                                 links.Each(usagesFiller.AddFirstAndReturnConstant,
                                  ↪ usagesAsSourceQuery);
675                                 for (; i < usagesAsSourceCount; i++)
676                                 {
677                                     var usage = usages[i];
678                                     if (!equalityComparer.Equals(usage, oldLinkIndex))
679                                     {
680                                         links.Update(usage, newLinkIndex, links.GetTarget(usage));
681                                     }
682                                 }
683                             }
684                             if (usagesAsTargetCount > 0)
685                             {
686                                 links.Each(usagesFiller.AddFirstAndReturnConstant,
                                  ↪ usagesAsTargetQuery);
687                                 for (; i < usages.Length; i++)
688                                 {
689                                     var usage = usages[i];
690                                     if (!equalityComparer.Equals(usage, oldLinkIndex))
691                                     {
692                                         links.Update(usage, links.GetSource(usage), newLinkIndex);
693                                     }
694                                 }
695                             }
696                             ArrayPool.Free(usages);
697                         }
698                     }
699                 }
700                 return newLinkIndex;
701             }
702
703             /// <summary>
704             /// Replace one link with another (replaced link is deleted, children are updated or
                  ↪ deleted).
705             /// </summary>
706             [MethodImpl(MethodImplOptions.AggressiveInlining)]
707             public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
                  ↪ TLink newLinkIndex)
708             {
709                 var equalityComparer = EqualityComparer<TLink>.Default;
710                 if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
711                 {
712                     links.MergeUsages(oldLinkIndex, newLinkIndex);
713                     links.Delete(oldLinkIndex);
714                 }
715                 return newLinkIndex;
716             }
717
718             public static ILinks<TLink>
                  ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
719             {
720                 links = new LinksCascadeUsagesResolver<TLink>(links);
721                 links = new NonNullContentsLinkDeletionResolver<TLink>(links);
722                 links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
723                 return links;
724             }
725         }
726 }
```

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

```
1   using System.Collections.Generic;
2   using Platform.Interfaces;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Incrementers
7   {
8       public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
```

```
 9        {
10            private static readonly EqualityComparer<TLink> _equalityComparer =
              ↪  EqualityComparer<TLink>.Default;
11
12            private readonly TLink _frequencyMarker;
13            private readonly TLink _unaryOne;
14            private readonly IIncrementer<TLink> _unaryNumberIncrementer;
15
16            public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
              ↪  IIncrementer<TLink> unaryNumberIncrementer)
17                : base(links)
18            {
19                _frequencyMarker = frequencyMarker;
20                _unaryOne = unaryOne;
21                _unaryNumberIncrementer = unaryNumberIncrementer;
22            }
23
24            public TLink Increment(TLink frequency)
25            {
26                if (_equalityComparer.Equals(frequency, default))
27                {
28                    return Links.GetOrCreate(_unaryOne, _frequencyMarker);
29                }
30                var source = Links.GetSource(frequency);
31                var incrementedSource = _unaryNumberIncrementer.Increment(source);
32                return Links.GetOrCreate(incrementedSource, _frequencyMarker);
33            }
34        }
35    }
```

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

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

./Platform.Data.Doublets/ISynchronizedLinks.cs

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

```csharp
1   using Platform.Collections.Lists;
2   using Platform.Exceptions;
3   using Platform.Ranges;
4   using Platform.Singletons;
5   using System;
6   using System.Collections;
7   using System.Collections.Generic;
8   using System.Runtime.CompilerServices;
9
10  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12  namespace Platform.Data.Doublets
13  {
14      /// <summary>
15      /// Структура описывающая уникальную связь.
16      /// </summary>
17      public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
18      {
19          public static readonly Link<TLink> Null = new Link<TLink>();
20
21          private static readonly LinksConstants<TLink> _constants =
22          ↪  Default<LinksConstants<TLink>>.Instance;
22          private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
23
24          private const int Length = 3;
25
26          public readonly TLink Index;
27          public readonly TLink Source;
28          public readonly TLink Target;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
           ↪  Target);
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public Link(IList<TLink> values) => SetValues(values, out Index, out Source, out Target);
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          public Link(object other)
38          {
39              if (other is Link<TLink> otherLink)
40              {
41                  SetValues(ref otherLink, out Index, out Source, out Target);
42              }
43              else if(other is IList<TLink> otherList)
44              {
45                  SetValues(otherList, out Index, out Source, out Target);
46              }
47              else
48              {
49                  throw new NotSupportedException();
50              }
51          }
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
           ↪  Target);
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          public Link(TLink index, TLink source, TLink target)
58          {
59              Index = index;
60              Source = source;
61              Target = target;
62          }
63
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
           ↪  out TLink target)
66          {
67              index = other.Index;
68              source = other.Source;
69              target = other.Target;
70          }
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          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)]
                public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
                 ↪   ToString(Source, Target) : ToString(Index, Source, Target);

                #region IList

                public int Count => Length;

                public bool IsReadOnly => true;

                public TLink this[int index]
                {
                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
                    get
                    {
                        Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
                         ↪   nameof(index));
                        if (index == _constants.IndexPart)
                        {
                            return Index;
                        }
                        if (index == _constants.SourcePart)
                        {
```

```csharp
                    return Source;
                }
                if (index == _constants.TargetPart)
                {
                    return Target;
                }
                throw new NotSupportedException(); // Impossible path due to
                ↪ Ensure.ArgumentInRange
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => throw new NotSupportedException();
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerator<TLink> GetEnumerator()
        {
            yield return Index;
            yield return Source;
            yield return Target;
        }

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CopyTo(TLink[] array, int arrayIndex)
        {
            Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
            Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
            ↪ nameof(arrayIndex));
            if (arrayIndex + Length > array.Length)
            {
                throw new InvalidOperationException();
            }
            array[arrayIndex++] = Index;
            array[arrayIndex++] = Source;
            array[arrayIndex] = Target;
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int IndexOf(TLink item)
        {
            if (_equalityComparer.Equals(Index, item))
            {
                return _constants.IndexPart;
            }
            if (_equalityComparer.Equals(Source, item))
            {
                return _constants.SourcePart;
            }
            if (_equalityComparer.Equals(Target, item))
            {
                return _constants.TargetPart;
            }
            return -1;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Insert(int index, TLink item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void RemoveAt(int index) => throw new NotSupportedException();

        #endregion
    }
}
```

## ./Platform.Data.Doublets/LinkExtensions.cs

```csharp
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets
4  {
5      public static class 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

```csharp
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

```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.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

```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.Numbers.Raw
7  {
8      public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
9      {
10         public TLink Convert(TLink source) => (Integer<TLink>)new
             ↪  Hybrid<TLink>(source).AbsoluteValue;
11     }
12  }
```

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

```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 }
```

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

```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

```csharp
1  using System.Collections.Generic;
2  using Platform.Exceptions;
3  using Platform.Interfaces;
4  using Platform.Ranges;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
10     public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<int, TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪ EqualityComparer<TLink>.Default;
13
```

```
14          private readonly TLink[] _unaryNumberPowersOf2;

16          public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
17          {
18              _unaryNumberPowersOf2 = new TLink[64];
19              _unaryNumberPowersOf2[0] = one;
20          }

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;

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

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;

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

17          public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
18              : base(links)
19          {
20              _unaryOne = unaryOne;
21              InitUnaryToUInt64();
22          }

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          }

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
27          public TLink Convert(TLink sourceNumber)
28          {
29              var nullConstant = Links.Constants.Null;
30              var source = sourceNumber;
31              var target = nullConstant;
32              if (!_equalityComparer.Equals(source, nullConstant))
33              {
34                  while (true)
35                  {
36                      if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
37                      {
38                          SetBit(ref target, powerOf2Index);
39                          break;
40                      }
41                      else
42                      {
43                          powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
44                          SetBit(ref target, powerOf2Index);
45                          source = Links.GetTarget(source);
46                      }
47                  }
48              }
49              return target;
50          }
51
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
53        private static void SetBit(ref TLink target, int powerOf2Index) => target =
   ↪    Bit.Or(target, Bit.ShiftLeft(Integer<TLink>.One, powerOf2Index));
54    }
55  }
```

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

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

```
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 LinksAvlBalancedTreeMethodsBase<TLink> :
   ↪    SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
14     {
15         protected readonly TLink Break;
16         protected readonly TLink Continue;
17         protected readonly byte* Links;
18         protected readonly byte* Header;
19
20         public LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
   ↪    byte* header)
21         {
22             Links = links;
23             Header = header;
24             Break = constants.Break;
25             Continue = constants.Continue;
26         }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected abstract TLink GetTreeRoot();
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected abstract TLink GetBasePartValue(TLink link);
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
     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
         {
             //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
             return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
         }
     }

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

     [MethodImpl(MethodImplOptions.AggressiveInlining)]
     protected virtual bool GetRightIsChildValue(TLink value)
     {
         unchecked
         {
             //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
             return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
         }
```

```csharp
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
```

```csharp
                        {
                            current = GetRightOrDefault(current);
                        }
                    }
                    if (!EqualToZero(first))
                    {
                        current = first;
                        while (true)
                        {
                            if (IsEquals(handler(GetLinkValues(current)), Break))
                            {
                                return Break;
                            }
                            current = GetNext(current);
                            if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
                            {
                                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/LinksSizeBalancedTreeMethodsBase.cs

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

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

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> :
        SizeBalancedTreeMethods2<TLink>, ILinksTreeMethods<TLink>
    {
        protected readonly TLink Break;
        protected readonly TLink Continue;
        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)]
```

```csharp
        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))
                    {
                        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;
```

```
112              if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
     ↪  node.Key < root.Key
113              {
114                  root = GetLeftOrDefault(root);
115              }
116              else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
     ↪  node.Key > root.Key
117              {
118                  root = GetRightOrDefault(root);
119              }
120              else // node.Key == root.Key
121              {
122                  return root;
123              }
124          }
125          return Zero;
126      }
127
128      // TODO: Return indices range instead of references count
129      public TLink CountUsages(TLink link)
130      {
131          var root = GetTreeRoot();
132          var total = GetSize(root);
133          var totalRightIgnore = Zero;
134          while (!EqualToZero(root))
135          {
136              var @base = GetBasePartValue(root);
137              if (LessOrEqualThan(@base, link))
138              {
139                  root = GetRightOrDefault(root);
140              }
141              else
142              {
143                  totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
144                  root = GetLeftOrDefault(root);
145              }
146          }
147          root = GetTreeRoot();
148          var totalLeftIgnore = Zero;
149          while (!EqualToZero(root))
150          {
151              var @base = GetBasePartValue(root);
152              if (GreaterOrEqualThan(@base, link))
153              {
154                  root = GetLeftOrDefault(root);
155              }
156              else
157              {
158                  totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
159
160                  root = GetRightOrDefault(root);
161              }
162          }
163          return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
164      }
165
166      [MethodImpl(MethodImplOptions.AggressiveInlining)]
167      public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
     ↪  EachUsageCore(@base, GetTreeRoot(), handler);
168
169      // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
     ↪  low-level MSIL stack.
170      private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
171      {
172          var @continue = Continue;
173          if (EqualToZero(link))
174          {
175              return @continue;
176          }
177          var linkBasePart = GetBasePartValue(link);
178          var @break = Break;
179          if (GreaterThan(linkBasePart, @base))
180          {
181              if (IsEquals(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
182              {
183                  return @break;
184              }
185          }
186          else if (LessThan(linkBasePart, @base))
```

```
187                {
188                    if (IsEquals(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
189                    {
190                        return @break;
191                    }
192                }
193                else //if (linkBasePart == @base)
194                {
195                    if (IsEquals(handler(GetLinkValues(link)), @break))
196                    {
197                        return @break;
198                    }
199                    if (IsEquals(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
200                    {
201                        return @break;
202                    }
203                    if (IsEquals(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
204                    {
205                        return @break;
206                    }
207                }
208                return @continue;
209            }
210
211            protected override void PrintNodeValue(TLink node, StringBuilder sb)
212            {
213                ref var link = ref GetLinkReference(node);
214                sb.Append(' ');
215                sb.Append(link.Source);
216                sb.Append('-');
217                sb.Append('>');
218                sb.Append(link.Target);
219            }
220        }
221    }
```

./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesAvlBalancedTreeMethods.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 LinksSourcesAvlBalancedTreeMethods<TLink> :
        ↪ LinksAvlBalancedTreeMethodsBase<TLink>
8       {
9           public LinksSourcesAvlBalancedTreeMethods(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).LeftAsSource;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected unsafe override ref TLink GetRightReference(TLink node) => ref
            ↪ GetLinkReference(node).RightAsSource;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override void SetLeft(TLink node, TLink left) =>
            ↪ GetLinkReference(node).LeftAsSource = left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(TLink node, TLink right) =>
            ↪ GetLinkReference(node).RightAsSource = right;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override TLink GetSize(TLink node) =>
            ↪ GetSizeValue(GetLinkReference(node).SizeAsSource);
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
            ↪ GetLinkReference(node).SizeAsSource, size);
34
```

```csharp
        [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;
        }
    }
}
```

./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesSizeBalancedTreeMethods.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 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)]
```

```csharp
                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;
                }
            }
        }
```

./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsAvlBalancedTreeMethods.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 LinksTargetsAvlBalancedTreeMethods<TLink> :
        LinksAvlBalancedTreeMethodsBase<TLink>
    {
        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);
```

```csharp
31
32            [MethodImpl(MethodImplOptions.AggressiveInlining)]
33            protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
     ↪    GetLinkReference(node).SizeAsTarget, size);
34
35            [MethodImpl(MethodImplOptions.AggressiveInlining)]
36            protected override bool GetLeftIsChild(TLink node) =>
     ↪    GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);
37
38            [MethodImpl(MethodImplOptions.AggressiveInlining)]
39            protected override void SetLeftIsChild(TLink node, bool value) =>
     ↪    SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
40
41            [MethodImpl(MethodImplOptions.AggressiveInlining)]
42            protected override bool GetRightIsChild(TLink node) =>
     ↪    GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);
43
44            [MethodImpl(MethodImplOptions.AggressiveInlining)]
45            protected override void SetRightIsChild(TLink node, bool value) =>
     ↪    SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
46
47            [MethodImpl(MethodImplOptions.AggressiveInlining)]
48            protected override sbyte GetBalance(TLink node) =>
     ↪    GetBalanceValue(GetLinkReference(node).SizeAsTarget);
49
50            [MethodImpl(MethodImplOptions.AggressiveInlining)]
51            protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
     ↪    GetLinkReference(node).SizeAsTarget, value);
52
53            [MethodImpl(MethodImplOptions.AggressiveInlining)]
54            protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;
55
56            [MethodImpl(MethodImplOptions.AggressiveInlining)]
57            protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
58
59            [MethodImpl(MethodImplOptions.AggressiveInlining)]
60            protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
     ↪    TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
     ↪    IsEquals(firstTarget, secondTarget) && LessThan(firstSource, secondSource);
61
62            [MethodImpl(MethodImplOptions.AggressiveInlining)]
63            protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
     ↪    TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
     ↪    IsEquals(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);
64
65            [MethodImpl(MethodImplOptions.AggressiveInlining)]
66            protected override void ClearNode(TLink node)
67            {
68                ref var link = ref GetLinkReference(node);
69                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)]
```

```csharp
        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) => GetLinkReference(node).SizeAsTarget;

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

        [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;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

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

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

```csharp
        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)
        {
            _memory = memory;
            _memoryReservationStep = memoryReservationStep;
            Constants = 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);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Count(IList<TLink> restrictions)
        {
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
            if (restrictions.Count == 0)
            {
                return Total;
            }
            var constants = Constants;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Total;
                }
                return Exists(index) ? GetOne() : GetZero();
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Total; // Any - как отсутствие ограничения
                    }
                    return Add(SourcesTreeMethods.CountUsages(value),
                       TargetsTreeMethods.CountUsages(value));
                }
```

```
105             else
106             {
107                 if (!Exists(index))
108                 {
109                     return GetZero();
110                 }
111                 if (AreEqual(value, any))
112                 {
113                     return GetOne();
114                 }
115                 ref var storedLinkValue = ref GetLinkReference(index);
116                 if (AreEqual(storedLinkValue.Source, value) ||
                    ↪  AreEqual(storedLinkValue.Target, value))
117                 {
118                     return GetOne();
119                 }
120                 return GetZero();
121             }
122         }
123         if (restrictions.Count == 3)
124         {
125             var source = restrictions[constants.SourcePart];
126             var target = restrictions[constants.TargetPart];
127             if (AreEqual(index, any))
128             {
129                 if (AreEqual(source, any) && AreEqual(target, any))
130                 {
131                     return Total;
132                 }
133                 else if (AreEqual(source, any))
134                 {
135                     return TargetsTreeMethods.CountUsages(target);
136                 }
137                 else if (AreEqual(target, any))
138                 {
139                     return SourcesTreeMethods.CountUsages(source);
140                 }
141                 else //if(source != Any && target != Any)
142                 {
143                     // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
144                     var link = SourcesTreeMethods.Search(source, target);
145                     return AreEqual(link, constants.Null) ? GetZero() : GetOne();
146                 }
147             }
148             else
149             {
150                 if (!Exists(index))
151                 {
152                     return GetZero();
153                 }
154                 if (AreEqual(source, any) && AreEqual(target, any))
155                 {
156                     return GetOne();
157                 }
158                 ref var storedLinkValue = ref GetLinkReference(index);
159                 if (!AreEqual(source, any) && !AreEqual(target, any))
160                 {
161                     if (AreEqual(storedLinkValue.Source, source) &&
                        ↪  AreEqual(storedLinkValue.Target, target))
162                     {
163                         return GetOne();
164                     }
165                     return GetZero();
166                 }
167                 var value = default(TLink);
168                 if (AreEqual(source, any))
169                 {
170                     value = target;
171                 }
172                 if (AreEqual(target, any))
173                 {
174                     value = source;
175                 }
176                 if (AreEqual(storedLinkValue.Source, value) ||
                    ↪  AreEqual(storedLinkValue.Target, value))
177                 {
178                     return GetOne();
179                 }
```

```
180             return GetZero();
181         }
182     }
183     throw new NotSupportedException("Другие размеры и способы ограничений не
    ↪   поддерживаются.");
184 }
185
186 [MethodImpl(MethodImplOptions.AggressiveInlining)]
187 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
188 {
189     var constants = Constants;
190     var @break = constants.Break;
191     if (restrictions.Count == 0)
192     {
193         for (var link = GetOne(); LessOrEqualThan(link,
        ↪   GetHeaderReference().AllocatedLinks); link = Increment(link))
194         {
195             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
196             {
197                 return @break;
198             }
199         }
200         return @break;
201     }
202     var @continue = constants.Continue;
203     var any = constants.Any;
204     var index = restrictions[constants.IndexPart];
205     if (restrictions.Count == 1)
206     {
207         if (AreEqual(index, any))
208         {
209             return Each(handler, GetEmptyList());
210         }
211         if (!Exists(index))
212         {
213             return @continue;
214         }
215         return handler(GetLinkStruct(index));
216     }
217     if (restrictions.Count == 2)
218     {
219         var value = restrictions[1];
220         if (AreEqual(index, any))
221         {
222             if (AreEqual(value, any))
223             {
224                 return Each(handler, GetEmptyList());
225             }
226             if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
227             {
228                 return @break;
229             }
230             return Each(handler, new Link<TLink>(index, any, value));
231         }
232         else
233         {
234             if (!Exists(index))
235             {
236                 return @continue;
237             }
238             if (AreEqual(value, any))
239             {
240                 return handler(GetLinkStruct(index));
241             }
242             ref var storedLinkValue = ref GetLinkReference(index);
243             if (AreEqual(storedLinkValue.Source, value) ||
244                 AreEqual(storedLinkValue.Target, value))
245             {
246                 return handler(GetLinkStruct(index));
247             }
248             return @continue;
249         }
250     }
251     if (restrictions.Count == 3)
252     {
253         var source = restrictions[constants.SourcePart];
254         var target = restrictions[constants.TargetPart];
255         if (AreEqual(index, any))
```

```
256                     {
257                         if (AreEqual(source, any) && AreEqual(target, any))
258                         {
259                             return Each(handler, GetEmptyList());
260                         }
261                         else if (AreEqual(source, any))
262                         {
263                             return TargetsTreeMethods.EachUsage(target, handler);
264                         }
265                         else if (AreEqual(target, any))
266                         {
267                             return SourcesTreeMethods.EachUsage(source, handler);
268                         }
269                         else //if(source != Any && target != Any)
270                         {
271                             var link = SourcesTreeMethods.Search(source, target);
272                             return AreEqual(link, constants.Null) ? @continue :
                              ↪   handler(GetLinkStruct(link));
273                         }
274                     }
275                     else
276                     {
277                         if (!Exists(index))
278                         {
279                             return @continue;
280                         }
281                         if (AreEqual(source, any) && AreEqual(target, any))
282                         {
283                             return handler(GetLinkStruct(index));
284                         }
285                         ref var storedLinkValue = ref GetLinkReference(index);
286                         if (!AreEqual(source, any) && !AreEqual(target, any))
287                         {
288                             if (AreEqual(storedLinkValue.Source, source) &&
289                                 AreEqual(storedLinkValue.Target, target))
290                             {
291                                 return handler(GetLinkStruct(index));
292                             }
293                             return @continue;
294                         }
295                         var value = default(TLink);
296                         if (AreEqual(source, any))
297                         {
298                             value = target;
299                         }
300                         if (AreEqual(target, any))
301                         {
302                             value = source;
303                         }
304                         if (AreEqual(storedLinkValue.Source, value) ||
305                             AreEqual(storedLinkValue.Target, value))
306                         {
307                             return handler(GetLinkStruct(index));
308                         }
309                         return @continue;
310                     }
311                 }
312             throw new NotSupportedException("Другие размеры и способы ограничений не
                  ↪   поддерживаются.");
313         }
314
315         /// <remarks>
316         /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
              ↪   в другом месте (но не в менеджере памяти, а в логике Links)
317         /// </remarks>
318         [MethodImpl(MethodImplOptions.AggressiveInlining)]
319         public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
320         {
321             var constants = Constants;
322             var @null = constants.Null;
323             var linkIndex = restrictions[constants.IndexPart];
324             ref var link = ref GetLinkReference(linkIndex);
325             ref var header = ref GetHeaderReference();
326             ref var firstAsSource = ref header.FirstAsSource;
327             ref var firstAsTarget = ref header.FirstAsTarget;
328             // Будет корректно работать только в том случае, если пространство выделенной связи
                  ↪   предварительно заполнено нулями
329             if (!AreEqual(link.Source, @null))
```

```csharp
                {
                    SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
                }
                if (!AreEqual(link.Target, @null))
                {
                    TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
                }
                link.Source = substitution[constants.SourcePart];
                link.Target = substitution[constants.TargetPart];
                if (!AreEqual(link.Source, @null))
                {
                    SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
                }
                if (!AreEqual(link.Target, @null))
                {
                    TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
                }
                return linkIndex;
            }

            /// <remarks>
            /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
            ↪   пространство
            /// </remarks>
            public virtual TLink Create(IList<TLink> restrictions)
            {
                ref var header = ref GetHeaderReference();
                var freeLink = header.FirstFreeLink;
                if (!AreEqual(freeLink, Constants.Null))
                {
                    UnusedLinksListMethods.Detach(freeLink);
                }
                else
                {
                    var maximumPossibleInnerReference =
                    ↪   Constants.PossibleInnerReferencesRange.Maximum;
                    if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
                    {
                        throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
                    }
                    if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
                    {
                        _memory.ReservedCapacity += _memoryReservationStep;
                        SetPointers(_memory);
                        header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
                        ↪   LinkSizeInBytes);
                    }
                    header.AllocatedLinks = Increment(header.AllocatedLinks);
                    _memory.UsedCapacity += LinkSizeInBytes;
                    freeLink = header.AllocatedLinks;
                }
                return freeLink;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public virtual void Delete(IList<TLink> restrictions)
            {
                ref var header = ref GetHeaderReference();
                var link = restrictions[Constants.IndexPart];
                if (LessThan(link, header.AllocatedLinks))
                {
                    UnusedLinksListMethods.AttachAsFirst(link);
                }
                else if (AreEqual(link, header.AllocatedLinks))
                {
                    header.AllocatedLinks = Decrement(header.AllocatedLinks);
                    _memory.UsedCapacity -= LinkSizeInBytes;
                    // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪   пока не дойдём до первой существующей связи
                    // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
                    while (GreaterThan(header.AllocatedLinks, GetZero()) &&
                    ↪   IsUnusedLink(header.AllocatedLinks))
                    {
                        UnusedLinksListMethods.Detach(header.AllocatedLinks);
                        header.AllocatedLinks = Decrement(header.AllocatedLinks);
                        _memory.UsedCapacity -= LinkSizeInBytes;
                    }
                }
```

```csharp
403            }
404
405            [MethodImpl(MethodImplOptions.AggressiveInlining)]
406            public IList<TLink> GetLinkStruct(TLink linkIndex)
407            {
408                ref var link = ref GetLinkReference(linkIndex);
409                return new Link<TLink>(linkIndex, link.Source, link.Target);
410            }
411
412            /// <remarks>
413            /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
414                →  адрес реально поменялся
415            ///
416            /// Указатель this.links может быть в том же месте,
417            /// так как 0-я связь не используется и имеет такой же размер как Header,
418            /// поэтому header размещается в том же месте, что и 0-я связь
419            /// </remarks>
420            [MethodImpl(MethodImplOptions.AggressiveInlining)]
421            protected abstract void SetPointers(IResizableDirectMemory memory);
422
423            [MethodImpl(MethodImplOptions.AggressiveInlining)]
424            protected virtual void ResetPointers()
425            {
426                SourcesTreeMethods = null;
427                TargetsTreeMethods = null;
428                UnusedLinksListMethods = null;
429            }
430
431            [MethodImpl(MethodImplOptions.AggressiveInlining)]
432            protected abstract ref LinksHeader<TLink> GetHeaderReference();
433
434            [MethodImpl(MethodImplOptions.AggressiveInlining)]
435            protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
436
437            [MethodImpl(MethodImplOptions.AggressiveInlining)]
438            protected virtual bool Exists(TLink link)
439                => GreaterOrEqualThan(link, Constants.PossibleInnerReferencesRange.Minimum)
440                && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
441                && !IsUnusedLink(link);
442
443            [MethodImpl(MethodImplOptions.AggressiveInlining)]
444            protected virtual bool IsUnusedLink(TLink linkIndex)
445            {
446                if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
447                    →  is not needed
448                {
449                    ref var link = ref GetLinkReference(linkIndex);
450                    return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
451                }
452                else
453                {
454                    return true;
455                }
456            }
457
458            [MethodImpl(MethodImplOptions.AggressiveInlining)]
459            protected virtual TLink GetOne() => Integer<TLink>.One;
460
461            [MethodImpl(MethodImplOptions.AggressiveInlining)]
462            protected virtual TLink GetZero() => Integer<TLink>.Zero;
463
464            [MethodImpl(MethodImplOptions.AggressiveInlining)]
465            protected virtual bool AreEqual(TLink first, TLink second) =>
466                →  EqualityComparer.Equals(first, second);
467
468            [MethodImpl(MethodImplOptions.AggressiveInlining)]
469            protected virtual bool LessThan(TLink first, TLink second) => Comparer.Compare(first,
470                →  second) < 0;
471
472            [MethodImpl(MethodImplOptions.AggressiveInlining)]
473            protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
474                →  Comparer.Compare(first, second) <= 0;
475
476            [MethodImpl(MethodImplOptions.AggressiveInlining)]
477            protected virtual bool GreaterThan(TLink first, TLink second) => Comparer.Compare(first,
478                →  second) > 0;
479
480            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
475        protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
       ↪  Comparer.Compare(first, second) >= 0;
476
477        [MethodImpl(MethodImplOptions.AggressiveInlining)]
478        protected virtual long ConvertToUInt64(TLink value) => (Integer<TLink>)value;
479
480        [MethodImpl(MethodImplOptions.AggressiveInlining)]
481        protected virtual TLink ConvertToAddress(long value) => (Integer<TLink>)value;
482
483        [MethodImpl(MethodImplOptions.AggressiveInlining)]
484        protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
       ↪  second);
485
486        [MethodImpl(MethodImplOptions.AggressiveInlining)]
487        protected virtual TLink Subtract(TLink first, TLink second) =>
       ↪  Arithmetic<TLink>.Subtract(first, second);
488
489        [MethodImpl(MethodImplOptions.AggressiveInlining)]
490        protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
491
492        [MethodImpl(MethodImplOptions.AggressiveInlining)]
493        protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
494
495        [MethodImpl(MethodImplOptions.AggressiveInlining)]
496        protected virtual IList<TLink> GetEmptyList() => ArrayPool<TLink>.Empty;
497
498        #region Disposable
499
500        protected override bool AllowMultipleDisposeCalls => true;
501
502        protected override void Dispose(bool manual, bool wasDisposed)
503        {
504            if (!wasDisposed)
505            {
506                ResetPointers();
507                _memory.DisposeIfPossible();
508            }
509        }
510
511        #endregion
512    }
513 }
```

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

```csharp
1  using System.Runtime.CompilerServices;
2  using Platform.Numbers;
3  using Platform.Memory;
4  using static System.Runtime.CompilerServices.Unsafe;
5  using System;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
10 {
11     public unsafe partial class ResizableDirectMemoryLinks<TLink> :
       ↪  ResizableDirectMemoryLinksBase<TLink>
12     {
13         private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
14         private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
15         private byte* _header;
16         private byte* _links;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public ResizableDirectMemoryLinks(string address) : this(address, DefaultLinksSizeStep)
       ↪  { }
20
21         /// <summary>
22         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
       ↪  минимальным шагом расширения базы данных.
23         /// </summary>
24         /// <param name="address">Полный пусь к файлу базы данных.</param>
25         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
       ↪  6айтах.</param>
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public ResizableDirectMemoryLinks(string address, long memoryReservationStep) : this(new
       ↪  FileMappedResizableDirectMemory(address, memoryReservationStep),
       ↪  memoryReservationStep) { }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
        ↪  memoryReservationStep) : this(memory, memoryReservationStep, true) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
        ↪  memoryReservationStep, bool useAvlBasedIndex) : base(memory, memoryReservationStep)
        {
            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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<TLink> GetHeaderReference() => ref
        ↪  AsRef<LinksHeader<TLink>>(_header);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
        ↪  AsRef<RawLink<TLink>>(_links + LinkSizeInBytes * (Integer<TLink>)linkIndex);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
    ↪  ILinksListMethods<TLink>
    {
        private readonly byte* _links;
        private readonly byte* _header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnusedLinksListMethods(byte* links, byte* header)
        {
            _links = links;
            _header = header;
        }
```

```csharp
        [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 override TLink GetFirst() => GetHeaderReference().FirstFreeLink;

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

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

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

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

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

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

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

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

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

./Platform.Data.Doublets/ResizableDirectMemory/ILinksListMethods.cs
```csharp
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

./Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.cs
```csharp
using System;
using System.Collections.Generic;

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

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

./Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs
```csharp
using Platform.Unsafe;

#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

```
 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

```
 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
 6
 7   namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
 8   {
 9       public unsafe abstract class UInt64LinksAvlBalancedTreeMethodsBase :
          ↪  LinksAvlBalancedTreeMethodsBase<ulong>
10      {
11           protected new readonly RawLink<ulong>* Links;
12           protected new readonly LinksHeader<ulong>* Header;
13
14           public UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
              ↪  RawLink<ulong>* links, LinksHeader<ulong>* header)
15               : base(constants, (byte*)links, (byte*)header)
16           {
17               Links = links;
18               Header = header;
19           }
20
21           [MethodImpl(MethodImplOptions.AggressiveInlining)]
22           protected override ulong GetZero() => 0UL;
23
24           [MethodImpl(MethodImplOptions.AggressiveInlining)]
25           protected override bool EqualToZero(ulong value) => value == 0UL;
26
27           [MethodImpl(MethodImplOptions.AggressiveInlining)]
28           protected override bool IsEquals(ulong first, ulong second) => first == second;
29
30           [MethodImpl(MethodImplOptions.AggressiveInlining)]
31           protected override bool GreaterThanZero(ulong value) => value > 0UL;
32
33           [MethodImpl(MethodImplOptions.AggressiveInlining)]
34           protected override bool GreaterThan(ulong first, ulong second) => first > second;
35
36           [MethodImpl(MethodImplOptions.AggressiveInlining)]
37           protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
38
```

```csharp
        [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 ulong GetSizeValue(ulong value) => unchecked((value & 4294967264UL)
        ↪  >> 5);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
        ↪  unchecked(storedValue & 31UL | (size & 134217727UL) << 5);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChildValue(ulong value) => unchecked((value & 16UL) >>
        ↪  4 == 1UL);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
        ↪  storedValue = unchecked(storedValue & 4294967279UL | (As<bool, byte>(ref value) &
        ↪  1UL) << 4);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChildValue(ulong value) => unchecked((value & 8UL) >>
        ↪  3 == 1UL);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
        ↪  storedValue = unchecked(storedValue & 4294967287UL | (As<bool, byte>(ref value) &
        ↪  1UL) << 3);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override bool GreaterThanZero(ulong value) => value > 0UL;
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override bool GreaterThan(ulong first, ulong second) => first > second;
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
     ↪    always true for ulong
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
     ↪    always >= 0 for ulong
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
     ↪    for ulong
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          protected override bool LessThan(ulong first, ulong second) => first < second;
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override ulong Increment(ulong value) => ++value;
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override ulong Decrement(ulong value) => --value;
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          protected override ulong Add(ulong first, ulong second) => first + second;
```

```csharp
        [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)
        ↪  { }

        [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)]
```

```
39      protected override void SetLeftIsChild(ulong node, bool value) =>
        ↪   SetLeftIsChildValue(ref Links[node].SizeAsSource, value);

40
41      [MethodImpl(MethodImplOptions.AggressiveInlining)]
42      protected override bool GetRightIsChild(ulong node) =>
        ↪   GetRightIsChildValue(Links[node].SizeAsSource);

43
44      [MethodImpl(MethodImplOptions.AggressiveInlining)]
45      protected override void SetRightIsChild(ulong node, bool value) =>
        ↪   SetRightIsChildValue(ref Links[node].SizeAsSource, value);

46
47      [MethodImpl(MethodImplOptions.AggressiveInlining)]
48      protected override sbyte GetBalance(ulong node) =>
        ↪   GetBalanceValue(Links[node].SizeAsSource);

49
50      [MethodImpl(MethodImplOptions.AggressiveInlining)]
51      protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
        ↪   Links[node].SizeAsSource, value);

52
53      [MethodImpl(MethodImplOptions.AggressiveInlining)]
54      protected override ulong GetTreeRoot() => Header->FirstAsSource;

55
56      [MethodImpl(MethodImplOptions.AggressiveInlining)]
57      protected override ulong GetBasePartValue(ulong link) => Links[link].Source;

58
59      [MethodImpl(MethodImplOptions.AggressiveInlining)]
60      protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
        ↪   ulong secondSource, ulong secondTarget)
61          => firstSource < secondSource || firstSource == secondSource && firstTarget <
            ↪   secondTarget;

62
63      [MethodImpl(MethodImplOptions.AggressiveInlining)]
64      protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
        ↪   ulong secondSource, ulong secondTarget)
65          => firstSource > secondSource || firstSource == secondSource && firstTarget >
            ↪   secondTarget;

66
67      [MethodImpl(MethodImplOptions.AggressiveInlining)]
68      protected override void ClearNode(ulong node)
69      {
70          ref var link = ref Links[node];
71          link.LeftAsSource = 0UL;
72          link.RightAsSource = 0UL;
73          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
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 UInt64LinksTargetsAvlBalancedTreeMethods :
        ↪    UInt64LinksAvlBalancedTreeMethodsBase
8       {
9           public UInt64LinksTargetsAvlBalancedTreeMethods(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) => GetSizeValue(Links[node].SizeAsTarget);
```

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

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

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

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

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

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

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

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

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

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

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

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

./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.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 UInt64LinksTargetsSizeBalancedTreeMethods :
    ↪  UInt64LinksSizeBalancedTreeMethodsBase
    {
        public UInt64LinksTargetsSizeBalancedTreeMethods(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;
```

```csharp
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

```csharp
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
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
10  {
11      public unsafe class UInt64ResizableDirectMemoryLinks : ResizableDirectMemoryLinksBase<ulong>
12      {
13          private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
14          private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
15          private LinksHeader<ulong>* _header;
16          private RawLink<ulong>* _links;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          public UInt64ResizableDirectMemoryLinks(string address) : this(address,
            DefaultLinksSizeStep) { }
20
21          /// <summary>
22          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            минимальным шагом расширения базы данных.
23          /// </summary>
24          /// <param name="address">Полный пусть к файлу базы данных.</param>
25          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            байтах.</param>
```

```csharp
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
   ↪    this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
   ↪    memoryReservationStep) { }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
   ↪    DefaultLinksSizeStep) { }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
   ↪    memoryReservationStep) : this(memory, memoryReservationStep, true) { }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
   ↪    memoryReservationStep, bool useAvlBasedIndex) : base(memory, memoryReservationStep)
37         {
38             if (useAvlBasedIndex)
39             {
40                 _createSourceTreeMethods = () => new
   ↪    UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
41                 _createTargetTreeMethods = () => new
   ↪    UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
42             }
43             else
44             {
45                 _createSourceTreeMethods = () => new
   ↪    UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
46                 _createTargetTreeMethods = () => new
   ↪    UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
47             }
48             Init(memory, memoryReservationStep);
49         }
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override void SetPointers(IResizableDirectMemory memory)
53         {
54             _header = (LinksHeader<ulong>*)memory.Pointer;
55             _links = (RawLink<ulong>*)memory.Pointer;
56             SourcesTreeMethods = _createSourceTreeMethods();
57             TargetsTreeMethods = _createTargetTreeMethods();
58             UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
59         }
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected override void ResetPointers()
63         {
64             base.ResetPointers();
65             _links = null;
66             _header = null;
67         }
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
71
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
   ↪    _links[linkIndex];
74
75         [MethodImpl(MethodImplOptions.AggressiveInlining)]
76         protected override bool AreEqual(ulong first, ulong second) => first == second;
77
78         [MethodImpl(MethodImplOptions.AggressiveInlining)]
79         protected override bool LessThan(ulong first, ulong second) => first < second;
80
81         [MethodImpl(MethodImplOptions.AggressiveInlining)]
82         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
83
84         [MethodImpl(MethodImplOptions.AggressiveInlining)]
85         protected override bool GreaterThan(ulong first, ulong second) => first > second;
86
87         [MethodImpl(MethodImplOptions.AggressiveInlining)]
88         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
89
90         [MethodImpl(MethodImplOptions.AggressiveInlining)]
91         protected override ulong GetZero() => 0UL;
92
93         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
                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

```csharp
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)
            : base((byte*)links, (byte*)header)
        {
            _links = links;
            _header = header;
        }

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

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

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

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

```csharp
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Sequences.Converters
{
```

```csharp
       public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
       {
           public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }

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

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

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

```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;
```

```csharp
26        private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
27        private readonly TLink _minFrequencyToCompress;
28        private readonly bool _doInitialFrequenciesIncrement;
29        private Doublet<TLink> _maxDoublet;
30        private LinkFrequency<TLink> _maxDoubletData;
31
32        private struct HalfDoublet
33        {
34            public TLink Element;
35            public LinkFrequency<TLink> DoubletData;
36
37            public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
38            {
39                Element = element;
40                DoubletData = doubletData;
41            }
42
43            public override string ToString() => $"{Element}: ({DoubletData})";
44        }
45
46        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
     ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
47            : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
48        {
49        }
50
51        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
     ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
     ↪  doInitialFrequenciesIncrement)
52            : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
     ↪  doInitialFrequenciesIncrement)
53        {
54        }
55
56        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
     ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
     ↪  minFrequencyToCompress, bool doInitialFrequenciesIncrement)
57            : base(links)
58        {
59            _baseConverter = baseConverter;
60            _doubletFrequenciesCache = doubletFrequenciesCache;
61            if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
62            {
63                minFrequencyToCompress = Integer<TLink>.One;
64            }
65            _minFrequencyToCompress = minFrequencyToCompress;
66            _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
67            ResetMaxDoublet();
68        }
69
70        public override TLink Convert(IList<TLink> source) =>
     ↪  _baseConverter.Convert(Compress(source));
71
72        /// <remarks>
73        /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
74        /// Faster version (doublets' frequencies dictionary is not recreated).
75        /// </remarks>
76        private IList<TLink> Compress(IList<TLink> sequence)
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;
```

```csharp
 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            }
127
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;
135                while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
136                {
137                    var maxDoubletSource = _maxDoublet.Source;
138                    var maxDoubletTarget = _maxDoublet.Target;
139                    if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
140                    {
141                        _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
142                    }
143                    var maxDoubletReplacementLink = _maxDoubletData.Link;
144                    oldLength--;
145                    var oldLengthMinusTwo = oldLength - 1;
146                    // Substitute all usages
147                    int w = 0, r = 0; // (r == read, w == write)
148                    for (; r < oldLength; r++)
149                    {
150                        if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                         ↪  _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
151                        {
152                            if (r > 0)
153                            {
154                                var previous = copy[w - 1].Element;
155                                copy[w - 1].DoubletData.DecrementFrequency();
156                                copy[w - 1].DoubletData =
                                 ↪  _doubletFrequenciesCache.IncrementFrequency(previous,
                                 ↪  maxDoubletReplacementLink);
157                            }
158                            if (r < oldLengthMinusTwo)
159                            {
160                                var next = copy[r + 2].Element;
161                                copy[r + 1].DoubletData.DecrementFrequency();
162                                copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma⌋
                                 ↪  xDoubletReplacementLink,
                                 ↪  next);
163                            }
164                            copy[w++].Element = maxDoubletReplacementLink;
165                            r++;
166                            newLength--;
167                        }
168                        else
```

```
169                         {
170                             copy[w++] = copy[r];
171                         }
172                     }
173                     if (w < newLength)
174                     {
175                         copy[w] = copy[r];
176                     }
177                     oldLength = newLength;
178                     ResetMaxDoublet();
179                     UpdateMaxDoublet(copy, newLength);
180                 }
181                 return newLength;
182             }
183
184             [MethodImpl(MethodImplOptions.AggressiveInlining)]
185             private void ResetMaxDoublet()
186             {
187                 _maxDoublet = new Doublet<TLink>();
188                 _maxDoubletData = new LinkFrequency<TLink>();
189             }
190
191             [MethodImpl(MethodImplOptions.AggressiveInlining)]
192             private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
193             {
194                 Doublet<TLink> doublet = default;
195                 for (var i = 1; i < length; i++)
196                 {
197                     doublet.Source = copy[i - 1].Element;
198                     doublet.Target = copy[i].Element;
199                     UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
200                 }
201             }
202
203             [MethodImpl(MethodImplOptions.AggressiveInlining)]
204             private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
205             {
206                 var frequency = data.Frequency;
207                 var maxFrequency = _maxDoubletData.Frequency;
208                 //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
209                 //   (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
210                 //   compression string data (and gives collisions quickly) */ _maxDoublet.Source +
211                 //   _maxDoublet.Target)))
                     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
                        (_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;
```

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

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

        private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;

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

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

```
79                             previousLevel = levels[w];
80                             w++;
81                         }
82                         if (i == length - 1)
83                         {
84                             sequence[w] = sequence[i];
85                             levels[w] = levels[i];
86                             w++;
87                         }
88                     }
89                 }
90                 length = w;
91             }
92             return links.GetOrCreate(sequence[0], sequence[1]);
93         }
94
95         private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink
            ↪  current, TLink next)
96         {
97             return _comparer.Compare(previous, next) > 0
98                 ? _comparer.Compare(previous, current) < 0 ? previous : current
99                 : _comparer.Compare(next, current) < 0 ? next : current;
100         }
101
102         private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
            ↪  _comparer.Compare(next, current) < 0 ? next : current;
103
104         private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
            ↪  => _comparer.Compare(previous, current) < 0 ? previous : current;
105     }
106 }
```

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

```
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Converters
7  {
8      public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<IList<TLink>>
9      {
10         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11
12         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
13
14         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
           ↪  IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
           ↪  => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
15
16         public IList<TLink> Convert(IList<TLink> sequence)
17         {
18             var levels = new TLink[sequence.Count];
19             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
20             for (var i = 1; i < sequence.Count - 1; i++)
21             {
22                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
23                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
24                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
25             }
26             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
               ↪  sequence[sequence.Count - 1]);
27             return levels;
28         }
29
30         public TLink GetFrequencyNumber(TLink source, TLink target) =>
           ↪  _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
31     }
32 }
```

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

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

```
 7      public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
   ↪  ICriterionMatcher<TLink>
 8      {
 9          public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
10          public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);
11      }
12  }
```

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

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

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

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

```
39                    }
40                }
41                var left = cursor;
42                var right = appendant;
43                while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
44                {
45                    right = Links.GetOrCreate(left, right);
46                    left = cursor;
47                }
48                return Links.GetOrCreate(left, right);
49            }
50        }
51    }
```

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

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

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

```
1    using System;
2    using System.Linq;
3    using System.Collections.Generic;
4    using Platform.Interfaces;
5    using Platform.Collections;
6    using Platform.Collections.Lists;
7    using Platform.Collections.Segments;
8    using Platform.Collections.Segments.Walkers;
9    using Platform.Singletons;
10   using Platform.Numbers;
11   using Platform.Data.Doublets.Unicode;
12
13   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15   namespace Platform.Data.Doublets.Sequences
16   {
17       public class DuplicateSegmentsProvider<TLink> :
             ↪ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
             ↪ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
18       {
19           private readonly ILinks<TLink> _links;
20           private readonly ILinks<TLink> _sequences;
21           private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
22           private BitString _visited;
23
24           private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
                 ↪ IList<TLink>>>
25           {
26               private readonly IListEqualityComparer<TLink> _listComparer;
27               public ItemEquilityComparer() => _listComparer =
                     ↪ Default<IListEqualityComparer<TLink>>.Instance;
28               public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
                     ↪ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
                     ↪ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
                     ↪ right.Value);
29               public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
                     ↪ (_listComparer.GetHashCode(pair.Key),
                     ↪ _listComparer.GetHashCode(pair.Value)).GetHashCode();
30           }
31
32           private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
33           {
34               private readonly IListComparer<TLink> _listComparer;
35
36               public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
```

```
37
38              public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
                 ↪ KeyValuePair<IList<TLink>, IList<TLink>> right)
39              {
40                  var intermediateResult = _listComparer.Compare(left.Key, right.Key);
41                  if (intermediateResult == 0)
42                  {
43                      intermediateResult = _listComparer.Compare(left.Value, right.Value);
44                  }
45                  return intermediateResult;
46              }
47          }
48
49          public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
50              : base(minimumStringSegmentLength: 2)
51          {
52              _links = links;
53              _sequences = sequences;
54          }
55
56          public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
57          {
58              _groups = new HashSet<KeyValuePair<IList<TLink>,
                 ↪ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
59              var count = _links.Count();
60              _visited = new BitString((long)(Integer<TLink>)count + 1);
61              _links.Each(link =>
62              {
63                  var linkIndex = _links.GetIndex(link);
64                  var linkBitIndex = (long)(Integer<TLink>)linkIndex;
65                  if (!_visited.Get(linkBitIndex))
66                  {
67                      var sequenceElements = new List<TLink>();
68                      var filler = new ListFiller<TLink, TLink>(sequenceElements,
                         ↪ _sequences.Constants.Break);
69                      _sequences.Each(filler.AddAllValuesAndReturnConstant, new
                         ↪ LinkAddress<TLink>(linkIndex));
70                      if (sequenceElements.Count > 2)
71                      {
72                          WalkAll(sequenceElements);
73                      }
74                  }
75                  return _links.Constants.Continue;
76              });
77              var resultList = _groups.ToList();
78              var comparer = Default<ItemComparer>.Instance;
79              resultList.Sort(comparer);
#if DEBUG
80
81              foreach (var item in resultList)
82              {
83                  PrintDuplicates(item);
84              }
#endif
85
86              return resultList;
87          }
88
89          protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
             ↪ length) => new Segment<TLink>(elements, offset, length);
90
91          protected override void OnDublicateFound(Segment<TLink> segment)
92          {
93              var duplicates = CollectDuplicatesForSegment(segment);
94              if (duplicates.Count > 1)
95              {
96                  _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                     ↪ duplicates));
97              }
98          }
99
100         private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
101         {
102             var duplicates = new List<TLink>();
103             var readAsElement = new HashSet<TLink>();
104             var restrictions = segment.ConvertToRestrictionsValues();
105             restrictions[0] = _sequences.Constants.Any;
106             _sequences.Each(sequence =>
107             {
108                 var sequenceIndex = sequence[_sequences.Constants.IndexPart];
```

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

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

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

```csharp
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          }
58
59          public void PrintFrequencies(IList<TLink> sequence)
60          {
61              for (var i = 1; i < sequence.Count; i++)
62              {
63                  PrintFrequency(sequence[i - 1], sequence[i]);
64              }
65          }
66
67          public void PrintFrequency(TLink source, TLink target)
68          {
69              var number = GetFrequency(source, target).Frequency;
70              Console.WriteLine("({0},{1}) - {2}", source, target, number);
71          }
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
75          {
76              if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
77              {
78                  data.IncrementFrequency();
79              }
80              else
81              {
82                  var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
83                  data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
84                  if (!_equalityComparer.Equals(link, default))
85                  {
86                      data.Frequency = Arithmetic.Add(data.Frequency,
                            ↪ _frequencyCounter.Count(link));
87                  }
88                  _doubletsCache.Add(doublet, data);
89              }
90              return data;
91          }
92
93          public void ValidateFrequencies()
94          {
95              foreach (var entry in _doubletsCache)
96              {
97                  var value = entry.Value;
98                  var linkIndex = value.Link;
```

```
99              if (!_equalityComparer.Equals(linkIndex, default))
100             {
101                 var frequency = value.Frequency;
102                 var count = _frequencyCounter.Count(linkIndex);
103                 // TODO: Why `frequency` always greater than `count` by 1?
104                 if (((_comparer.Compare(frequency, count) > 0) &&
                    ↪  (_comparer.Compare(Arithmetic.Subtract(frequency, count),
                    ↪  Integer<TLink>.One) > 0))
105                     || ((_comparer.Compare(count, frequency) > 0) &&
                    ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency),
                    ↪  Integer<TLink>.One) > 0)))
106                 {
107                     throw new InvalidOperationException("Frequencies validation failed.");
108                 }
109             }
110             //else
111             //{
112             //    if (value.Frequency > 0)
113             //    {
114             //        var frequency = value.Frequency;
115             //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
116             //        var count = _countLinkFrequency(linkIndex);
117
118             //        if ((frequency > count && frequency - count > 1) || (count > frequency
                    ↪  && count - frequency > 1))
119             //            throw new Exception("Frequencies validation failed.");
120             //    }
121             //}
122         }
123     }
124 }
125 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Numbers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
7  {
8      public class LinkFrequency<TLink>
9      {
10         public TLink Frequency { get; set; }
11         public TLink Link { get; set; }
12
13         public LinkFrequency(TLink frequency, TLink link)
14         {
15             Frequency = frequency;
16             Link = link;
17         }
18
19         public LinkFrequency() { }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);
26
27         public override string ToString() => $"F: {Frequency}, L: {Link}";
28     }
29 }
```

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

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

```
12          }
13      }
```

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

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

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

```
 1   using System.Collections.Generic;
 2   using Platform.Interfaces;
 3   using Platform.Numbers;
 4   using Platform.Data.Sequences;
 5
 6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 7
 8   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
 9   {
10       public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
11       {
12           private static readonly EqualityComparer<TLink> _equalityComparer =
       ↪  EqualityComparer<TLink>.Default;
13           private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
14
15           protected readonly ILinks<TLink> _links;
16           protected readonly TLink _sequenceLink;
17           protected readonly TLink _symbol;
18           protected TLink _total;
19
20           public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
       ↪  TLink symbol)
21           {
22               _links = links;
23               _sequenceLink = sequenceLink;
24               _symbol = symbol;
25               _total = default;
26           }
27
28           public virtual TLink Count()
29           {
30               if (_comparer.Compare(_total, default) > 0)
31               {
32                   return _total;
33               }
34               StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
       ↪  IsElement, VisitElement);
35               return _total;
36           }
37
38           private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
       ↪  _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
       ↪  IsPartialPoint
39
40           private bool VisitElement(TLink element)
41           {
42               if (_equalityComparer.Equals(element, _symbol))
43               {
```

```
44          _total = Arithmetic.Increment(_total);
45        }
46        return true;
47      }
48    }
49  }
```

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

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

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

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

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

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

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

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

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

```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.Sequences.HeightProviders
{
    public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
        ISequenceHeightProvider<TLink>
```

```csharp
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

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

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

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

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

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

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

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

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

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

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

```csharp
using Platform.Interfaces;

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

```

```
  5   namespace Platform.Data.Doublets.Sequences.HeightProviders
  6   {
  7       public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
  8       {
  9       }
 10   }
```

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

```
  1   using Platform.Collections;
  2   using System.Collections.Generic;
  3
  4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
  5
  6   namespace Platform.Data.Doublets.Sequences
  7   {
  8       public static class IListExtensions
  9       {
 10           public static TLink[] ExtractValues<TLink>(this IList<TLink> restrictions)
 11           {
 12               if(restrictions.IsNullOrEmpty() || restrictions.Count == 1)
 13               {
 14                   return new TLink[0];
 15               }
 16               var values = new TLink[restrictions.Count - 1];
 17               for (int i = 1, j = 0; i < restrictions.Count; i++, j++)
 18               {
 19                   values[j] = restrictions[i];
 20               }
 21               return values;
 22           }
 23
 24           public static IList<TLink> ConvertToRestrictionsValues<TLink>(this IList<TLink> list)
 25           {
 26               var restrictions = new TLink[list.Count + 1];
 27               for (int i = 0, j = 1; i < list.Count; i++, j++)
 28               {
 29                   restrictions[j] = list[i];
 30               }
 31               return restrictions;
 32           }
 33       }
 34   }
```

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

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

```
33                        return false;
34                    }
35                    var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
36                    if (indexed)
37                    {
38                        _cache.IncrementFrequency(source, target);
39                    }
40                    return indexed;
41                }
42
43            public bool MightContain(IList<TLink> sequence)
44            {
45                    var indexed = true;
46                    var i = sequence.Count;
47                    while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
48                    return indexed;
49            }
50
51            private bool IsIndexed(TLink source, TLink target)
52            {
53                    var frequency = _cache.GetFrequency(source, target);
54                    if (frequency == null)
55                    {
56                        return false;
57                    }
58                    return !_equalityComparer.Equals(frequency.Frequency, default);
59            }
60        }
61    }
```

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

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

```
46        {
47            var previousFrequency = _frequencyPropertyOperator.Get(link);
48            var frequency = _frequencyIncrementer.Increment(previousFrequency);
49            _frequencyPropertyOperator.Set(link, frequency);
50        }
51    }
52 }
```

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

```
1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Indexes
6  {
7      public interface ISequenceIndex<TLink>
8      {
9          /// <summary>
10         /// Индексирует последовательность глобально, и возвращает значение,
11         /// определяющие была ли запрошенная последовательность проиндексирована ранее.
12         /// </summary>
13         /// <param name="sequence">Последовательность для индексации.</param>
14         bool Add(IList<TLink> sequence);
15
16         bool MightContain(IList<TLink> sequence);
17     }
18 }
```

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

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

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

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

```
10
11          private readonly ISynchronizedLinks<TLink> _links;
12
13          public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
14
15          public bool Add(IList<TLink> sequence)
16          {
17              var indexed = true;
18              var i = sequence.Count;
19              var links = _links.Unsync;
20              _links.SyncRoot.ExecuteReadOperation(() =>
21              {
22                  while (--i >= 1 && (indexed =
                      ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                      ↪  sequence[i]), default))) { }
23              });
24              if (!indexed)
25              {
26                  _links.SyncRoot.ExecuteWriteOperation(() =>
27                  {
28                      for (; i >= 1; i--)
29                      {
30                          links.GetOrCreate(sequence[i - 1], sequence[i]);
31                      }
32                  });
33              }
34              return indexed;
35          }
36
37          public bool MightContain(IList<TLink> sequence)
38          {
39              var links = _links.Unsync;
40              return _links.SyncRoot.ExecuteReadOperation(() =>
41              {
42                  var indexed = true;
43                  var i = sequence.Count;
44                  while (--i >= 1 && (indexed =
                      ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                      ↪  sequence[i]), default))) { }
45                  return indexed;
46              });
47          }
48      }
49  }
```

./Platform.Data.Doublets/Sequences/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) { }
```

```csharp
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++)
56             {
57                 _list.Add(collection[i]);
58             }
59             return _returnConstant;
60         }
61     }
62 }
```

./Platform.Data.Doublets/Sequences/Sequences.cs
```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections;
6  using Platform.Collections.Lists;
7  using Platform.Threading.Synchronization;
8  using Platform.Singletons;
9  using LinkIndex = System.UInt64;
10 using Platform.Data.Doublets.Sequences.Walkers;
11 using Platform.Collections.Stacks;
12 using Platform.Collections.Arrays;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     /// <summary>
19     /// Представляет коллекцию последовательностей связей.
20     /// </summary>
21     /// <remarks>
22     /// Обязательно реализовать атомарность каждого публичного метода.
23     ///
24     /// TODO:
25     ///
26     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
27     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
28     ///   ↪ вместе, все числа вместе и т.п.
29     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
30     ///   ↪ графа)
31     ///
32     /// x*y - найти все связи между, в последовательностях любой формы, если не стоит
33     ///   ↪ ограничитель на то, что является последовательностью, а что нет,
34     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
35     ///   ↪ порядке.
```

```
32          ///
33          /// Рост последовательности слева и справа.
34          /// Поиск со звёздочкой.
35          /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
36          /// так же проблема может быть решена при реализации дистанционных триггеров.
37          /// Нужны ли уникальные указатели вообще?
38          /// Что если обращение к информации будет происходить через содержимое всегда?
39          ///
40          /// Писать тесты.
41          ///
42          ///
43          /// Можно убрать зависимость от конкретной реализации Links,
44          /// на зависимость от абстрактного элемента, который может быть представлен несколькими
        ↪  способами.
45          ///
46          /// Можно ли как-то сделать один общий интерфейс
47          ///
48          ///
49          /// Блокчейн и/или гит для распределённой записи транзакций.
50          ///
51          /// </remarks>
52          public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
        ↪  (после завершения реализации Sequences)
53          {
54              /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
                ↪  связей.</summary>
55              public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
56
57              public SequencesOptions<LinkIndex> Options { get; }
58              public SynchronizedLinks<LinkIndex> Links { get; }
59              private readonly ISynchronization _sync;
60
61              public LinksConstants<LinkIndex> Constants { get; }
62
63              public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
64              {
65                  Links = links;
66                  _sync = links.SyncRoot;
67                  Options = options;
68                  Options.ValidateOptions();
69                  Options.InitOptions(Links);
70                  Constants = Default<LinksConstants<LinkIndex>>.Instance;
71              }
72
73              public Sequences(SynchronizedLinks<LinkIndex> links)
74                  : this(links, new SequencesOptions<LinkIndex>())
75              {
76              }
77
78              public bool IsSequence(LinkIndex sequence)
79              {
80                  return _sync.ExecuteReadOperation(() =>
81                  {
82                      if (Options.UseSequenceMarker)
83                      {
84                          return Options.MarkedSequenceMatcher.IsMatched(sequence);
85                      }
86                      return !Links.Unsync.IsPartialPoint(sequence);
87                  });
88              }
89
90              [MethodImpl(MethodImplOptions.AggressiveInlining)]
91              private LinkIndex GetSequenceByElements(LinkIndex sequence)
92              {
93                  if (Options.UseSequenceMarker)
94                  {
95                      return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
96                  }
97                  return sequence;
98              }
99
100             private LinkIndex GetSequenceElements(LinkIndex sequence)
101             {
102                 if (Options.UseSequenceMarker)
103                 {
104                     var linkContents = new Link<ulong>(Links.GetLink(sequence));
105                     if (linkContents.Source == Options.SequenceMarkerLink)
106                     {
107                         return linkContents.Target;
```

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

        #region Count

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

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

        #endregion

        #region Create

        public LinkIndex Create(IList<LinkIndex> restrictions)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (restrictions.IsNullOrEmpty())
                {
                    return Constants.Null;
                }
                Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
                return CreateCore(restrictions);
            });
```

```csharp
        }

        private LinkIndex CreateCore(IList<LinkIndex> restrictions)
        {
            LinkIndex[] sequence = restrictions.ExtractValues();
            if (Options.UseIndex)
            {
                Options.Index.Add(sequence);
            }
            var sequenceRoot = default(LinkIndex);
            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
            {
                var matches = Each(restrictions);
                if (matches.Count > 0)
                {
                    sequenceRoot = matches[0];
                }
            }
            else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
            {
                return CompactCore(sequence);
            }
            if (sequenceRoot == default)
            {
                sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
            }
            if (Options.UseSequenceMarker)
            {
                Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
            }
            return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
        }

        #endregion

        #region Each

        public List<LinkIndex> Each(IList<LinkIndex> sequence)
        {
            var results = new List<LinkIndex>();
            var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
            Each(filler.AddFirstAndReturnConstant, sequence);
            return results;
        }

        public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
            restrictions)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (restrictions.IsNullOrEmpty())
                {
                    return Constants.Continue;
                }
                Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
                if (restrictions.Count == 1)
                {
                    var link = restrictions[0];
                    var any = Constants.Any;
                    if (link == any)
                    {
                        if (Options.UseSequenceMarker)
                        {
                            return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
                                Options.SequenceMarkerLink, any));
                        }
                        else
                        {
                            return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
                                any));
                        }
                    }
                    var sequence =
                        Options.Walker.Walk(link).ToArray().ConvertToRestrictionsValues();
                    sequence[0] = link;
                    return handler(sequence);
                }
                else if (restrictions.Count == 2)
```

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

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

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

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

        private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            right, LinkIndex stepFrom)
```

```
328             {
329                 var upStep = stepFrom;
330                 var firstSource = Links.Unsync.GetTarget(upStep);
331                 while (firstSource != right && firstSource != upStep)
332                 {
333                     upStep = firstSource;
334                     firstSource = Links.Unsync.GetSource(upStep);
335                 }
336                 if (firstSource == right)
337                 {
338                     return handler(new LinkAddress<LinkIndex>(stepFrom));
339                 }
340                 return Constants.Continue;
341             }
342
343             private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↪    LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
    ↪    leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
    ↪    right));
344
345             private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪    left, LinkIndex stepFrom)
346             {
347                 var upStep = stepFrom;
348                 var firstTarget = Links.Unsync.GetSource(upStep);
349                 while (firstTarget != left && firstTarget != upStep)
350                 {
351                     upStep = firstTarget;
352                     firstTarget = Links.Unsync.GetTarget(upStep);
353                 }
354                 if (firstTarget == left)
355                 {
356                     return handler(new LinkAddress<LinkIndex>(stepFrom));
357                 }
358                 return Constants.Continue;
359             }
360
361             #endregion
362
363             #region Update
364
365             public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
366             {
367                 var sequence = restrictions.ExtractValues();
368                 var newSequence = substitution.ExtractValues();
369
370                 if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
371                 {
372                     return Constants.Null;
373                 }
374                 if (sequence.IsNullOrEmpty())
375                 {
376                     return Create(substitution);
377                 }
378                 if (newSequence.IsNullOrEmpty())
379                 {
380                     Delete(restrictions);
381                     return Constants.Null;
382                 }
383                 return _sync.ExecuteWriteOperation(() =>
384                 {
385                     Links.EnsureEachLinkIsAnyOrExists(sequence);
386                     Links.EnsureEachLinkExists(newSequence);
387                     return UpdateCore(sequence, newSequence);
388                 });
389             }
390
391             private LinkIndex UpdateCore(LinkIndex[] sequence, LinkIndex[] newSequence)
392             {
393                 LinkIndex bestVariant;
394                 if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
    ↪    !sequence.EqualTo(newSequence))
395                 {
396                     bestVariant = CompactCore(newSequence);
397                 }
398                 else
399                 {
400                     bestVariant = CreateCore(newSequence);
401                 }
```

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

```csharp
        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);
                    if (Options.UseCascadeDelete || CountUsages(link) == 0)
                    {
                        if (sequenceLink != Constants.Null)
                        {
                            Links.Unsync.Delete(sequenceLink);
                        }
                        Links.Unsync.Delete(link);
                    }
                }
                else
                {
                    if (Options.UseCascadeDelete || CountUsages(link) == 0)
                    {
                        Links.Unsync.Delete(link);
                    }
                }
            }
        }

        #endregion

        #region Compactification

        /// <remarks>
        /// bestVariant можно выбирать по максимальному числу использований,
        /// но балансированный позволяет гарантировать уникальность (если есть возможность,
        /// гарантировать его использование в других местах).
        /// 
        /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
        /// </remarks>
        public LinkIndex Compact(params LinkIndex[] sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return Constants.Null;
                }
                Links.EnsureEachLinkExists(sequence);
                return CompactCore(sequence);
            });
        }

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

        #endregion

        #region Garbage Collection

        /// <remarks>
        /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
        ↪  определить извне или в унаследованном классе
```

```csharp
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
        ↪  !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;

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

        #endregion

        #region Walkers

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

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

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

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

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

            private bool FullMatchCore(LinkIndex element)
            {
                if (_filterPosition == _patternSequence.Count)
```

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

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

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

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

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

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

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

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

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

```csharp
        /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
        /// </remarks>
        public ulong[] CreateAllVariants2(ulong[] sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return new ulong[0];
                }
                Links.EnsureEachLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return sequence;
                }
                return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
            });
        }

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

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

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

        #endregion

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

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

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

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

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

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

```csharp
                    }
                    return true;
                });
                // |_x        ... x_o
                // |_o        |___|
                Links.Each(Constants.Any, sequence[0], doublet =>
                {
                    var match = Links.SearchOrDefault(doublet, sequence[1]);
                    if (match != 0)
                    {
                        handler(new LinkAddress<LinkIndex>(match));
                    }
                    return true;
                });
                //           ._x o_.
                //           |___|
                PartialStepRight(x => handler(x), sequence[0], sequence[1]);
            }
            else
            {
                throw new NotImplementedException();
            }
        }

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

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

        private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
        ↪  stepFrom)
        {
            var upStep = stepFrom;
            var firstSource = Links.Unsync.GetTarget(upStep);
            while (firstSource != right && firstSource != upStep)
            {
                upStep = firstSource;
                firstSource = Links.Unsync.GetSource(upStep);
            }
            if (firstSource == right)
            {
                handler(new LinkAddress<LinkIndex>(stepFrom));
            }
        }

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

        private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(Constants.Any, right, leftStep =>
```

```csharp
                {
                    TryStepLeftUp(handler, left, leftStep);
                    return true;
                });
        }

        private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
        {
            var upStep = stepFrom;
            var firstTarget = Links.Unsync.GetSource(upStep);
            while (firstTarget != left && firstTarget != upStep)
            {
                upStep = firstTarget;
                firstTarget = Links.Unsync.GetTarget(upStep);
            }
            if (firstTarget == left)
            {
                handler(new LinkAddress<LinkIndex>(stepFrom));
            }
        }

        private bool StartsWith(ulong sequence, ulong link)
        {
            var upStep = sequence;
            var firstSource = Links.Unsync.GetSource(upStep);
            while (firstSource != link && firstSource != upStep)
            {
                upStep = firstSource;
                firstSource = Links.Unsync.GetSource(upStep);
            }
            return firstSource == link;
        }

        private bool EndsWith(ulong sequence, ulong link)
        {
            var upStep = sequence;
            var lastTarget = Links.Unsync.GetTarget(upStep);
            while (lastTarget != link && lastTarget != upStep)
            {
                upStep = lastTarget;
                lastTarget = Links.Unsync.GetTarget(upStep);
            }
            return lastTarget == link;
        }

        public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new List<ulong>();
                if (sequence.Length > 0)
                {
                    Links.EnsureEachLinkExists(sequence);
                    var firstElement = sequence[0];
                    if (sequence.Length == 1)
                    {
                        results.Add(firstElement);
                        return results;
                    }
                    if (sequence.Length == 2)
                    {
                        var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
                        if (doublet != Constants.Null)
                        {
                            results.Add(doublet);
                        }
                        return results;
                    }
                    var linksInSequence = new HashSet<ulong>(sequence);
                    void handler(IList<LinkIndex> result)
                    {
                        var resultIndex = result[Links.Constants.IndexPart];
                        var filterPosition = 0;
                        StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
                            Links.Unsync.GetTarget,
                            x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                            x =>
                        {
                            if (filterPosition == sequence.Length)
```

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

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

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

```
481              });
482          }
483
484          public const int MaxSequenceFormatSize = 200;
485
486          public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
     ↪      => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
487
488          public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
     ↪      elementToString, bool insertComma, params LinkIndex[] knownElements) =>
     ↪      Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
     ↪      elementToString, insertComma, knownElements));
489
490          private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
     ↪      Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
     ↪      LinkIndex[] knownElements)
491          {
492              var linksInSequence = new HashSet<ulong>(knownElements);
493              //var entered = new HashSet<ulong>();
494              var sb = new StringBuilder();
495              sb.Append('{');
496              if (links.Exists(sequenceLink))
497              {
498                  StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
499                      x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
     ↪          entered.AddAndReturnVoid, x => { }, entered.DoNotContains
                      {
500                          if (insertComma && sb.Length > 1)
501                          {
502                              sb.Append(',');
503                          }
504                          //if (entered.Contains(element))
505                          //{
506                          //    sb.Append('{');
507                          //    elementToString(sb, element);
508                          //    sb.Append('}');
509                          //}
510                          //else
511                          elementToString(sb, element);
512                          if (sb.Length < MaxSequenceFormatSize)
513                          {
514                              return true;
515                          }
516                          sb.Append(insertComma ? ", ..." : "...");
517                          return false;
518                      });
519              }
520              sb.Append('}');
521              return sb.ToString();
522          }
523
524          public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
     ↪      knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
     ↪      knownElements);
525
526          public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
     ↪      LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
     ↪      Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
     ↪      sequenceLink, elementToString, insertComma, knownElements));
527
528          private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
     ↪      Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
     ↪      LinkIndex[] knownElements)
529          {
530              var linksInSequence = new HashSet<ulong>(knownElements);
531              var entered = new HashSet<ulong>();
532              var sb = new StringBuilder();
533              sb.Append('{');
534              if (links.Exists(sequenceLink))
535              {
536                  StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
537                      x => linksInSequence.Contains(x) || links.IsFullPoint(x),
     ↪          entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
                      {
538                          if (insertComma && sb.Length > 1)
539                          {
540                              sb.Append(',');
541                          }
542
```

```csharp
                }
                if (entered.Contains(element))
                {
                    sb.Append('{');
                    elementToString(sb, element);
                    sb.Append('}');
                }
                else
                {
                    elementToString(sb, element);
                }
                if (sb.Length < MaxSequenceFormatSize)
                {
                    return true;
                }
                sb.Append(insertComma ? ", ..." : "...");
                return false;
            });
        }
        sb.Append('}');
        return sb.ToString();
    }

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

```
620
621          public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
622          {
623              return _sync.ExecuteReadOperation(() =>
624              {
625                  if (sequence.Length > 0)
626                  {
627                      Links.EnsureEachLinkExists(sequence);
628                      var results = new HashSet<ulong>();
629                      for (var i = 0; i < sequence.Length; i++)
630                      {
631                          AllUsagesCore(sequence[i], results);
632                      }
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          }
641
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.EnsureEachLinkExists(sequence);
649
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          }
665
666          //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
667          //{
668          //     return Sync.ExecuteReadOperation(() =>
669          //     {
670          //         if (sequence.Length > 0)
671          //         {
672          //             _links.EnsureEachLinkIsAnyOrExists(sequence);
673
674          //             var firstResults = new HashSet<ulong>();
675          //             var lastResults = new HashSet<ulong>();
676
677          //             var first = sequence.First(x => x != LinksConstants.Any);
678          //             var last = sequence.Last(x => x != LinksConstants.Any);
679
680          //             AllUsagesCore(first, firstResults);
681          //             AllUsagesCore(last, lastResults);
682
683          //             firstResults.IntersectWith(lastResults);
684
685          //             //for (var i = 0; i < sequence.Length; i++)
686          //             //    AllUsagesCore(sequence[i], results);
687
688          //             var filteredResults = new HashSet<ulong>();
689          //             var matcher = new Matcher(this, sequence, filteredResults, null);
690          //             matcher.AddAllPartialMatchedToResults(firstResults);
691          //             return filteredResults;
692          //         }
693
694          //         return new HashSet<ulong>();
695          //     });
696          //}
697
```

```
698        public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
699        {
700            return _sync.ExecuteReadOperation(() =>
701            {
702                if (sequence.Length > 0)
703                {
704                    Links.EnsureEachLinkIsAnyOrExists(sequence);
705                    var firstResults = new HashSet<ulong>();
706                    var lastResults = new HashSet<ulong>();
707                    var first = sequence.First(x => x != Constants.Any);
708                    var last = sequence.Last(x => x != Constants.Any);
709                    AllUsagesCore(first, firstResults);
710                    AllUsagesCore(last, lastResults);
711                    firstResults.IntersectWith(lastResults);
712                    //for (var i = 0; i < sequence.Length; i++)
713                    //    AllUsagesCore(sequence[i], results);
714                    var filteredResults = new HashSet<ulong>();
715                    var matcher = new Matcher(this, sequence, filteredResults, null);
716                    matcher.AddAllPartialMatchedToResults(firstResults);
717                    return filteredResults;
718                }
719                return new HashSet<ulong>();
720            });
721        }
722
723        public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
           ↪ IList<ulong> sequence)
724        {
725            return _sync.ExecuteReadOperation(() =>
726            {
727                if (sequence.Count > 0)
728                {
729                    Links.EnsureEachLinkExists(sequence);
730                    var results = new HashSet<LinkIndex>();
731                    //var nextResults = new HashSet<ulong>();
732                    //for (var i = 0; i < sequence.Length; i++)
733                    //{
734                    //    AllUsagesCore(sequence[i], nextResults);
735                    //    if (results.IsNullOrEmpty())
736                    //    {
737                    //        results = nextResults;
738                    //        nextResults = new HashSet<ulong>();
739                    //    }
740                    //    else
741                    //    {
742                    //        results.IntersectWith(nextResults);
743                    //        nextResults.Clear();
744                    //    }
745                    //}
746                    var collector1 = new AllUsagesCollector1(Links.Unsync, results);
747                    collector1.Collect(Links.Unsync.GetLink(sequence[0]));
748                    var next = new HashSet<ulong>();
749                    for (var i = 1; i < sequence.Count; i++)
750                    {
751                        var collector = new AllUsagesCollector1(Links.Unsync, next);
752                        collector.Collect(Links.Unsync.GetLink(sequence[i]));
753
754                        results.IntersectWith(next);
755                        next.Clear();
756                    }
757                    var filteredResults = new HashSet<ulong>();
758                    var matcher = new Matcher(this, sequence, filteredResults, null,
                       ↪ readAsElements);
759                    matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
                       ↪ x)); // OrderBy is a Hack
760                    return filteredResults;
761                }
762                return new HashSet<ulong>();
763            });
764        }
765
766        // Does not work
767        //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
           ↪ params ulong[] sequence)
768        //{
769        //    var visited = new HashSet<ulong>();
770        //    var results = new HashSet<ulong>();
```

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

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

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

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

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

        private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
        ↪   usages)
        {
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```csharp
        }

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

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

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

```csharp
                {
                    AllUsagesCore(uniqueSequenceElement, results);
                }
                var filteredResults = new HashSet<ulong>();
                var matcher = new PatternMatcher(this, patternSequence, filteredResults);
                matcher.AddAllPatternMatchedToResults(results);
                return filteredResults;
            }
            return new HashSet<ulong>();
        });
    }

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

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

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

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

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

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

```csharp
                    newSequence[j++] = sequence[i];
                }
                return newSequence;
            }

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

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

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

            #region From Triplets

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

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

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

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

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

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

```csharp
            }

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

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

        #endregion

        #region Walkers

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

            #region Pattern Match

            enum PatternBlockType
            {
                Undefined,
                Gap,
                Elements
            }

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

```csharp
            }

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

            #endregion

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

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

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

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

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

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

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

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

```
1905                        {
1906                            _sequencePosition++;
1907                        }
1908                    }
1909                    return true;
1910                    //if (_patternSequence[_patternPosition] != element)
1911                    //    return false;
1912                    //else
1913                    //{
1914                    //    _sequencePosition++;
1915                    //    _patternPosition++;
1916                    //    return true;
1917                    //}
1918                    ////////
1919                    //if (_filterPosition == _patternSequence.Length)
1920                    //{
1921                    //    _filterPosition = -2; // Длиннее чем нужно
1922                    //    return false;
1923                    //}
1924                    //if (element != _patternSequence[_filterPosition])
1925                    //{
1926                    //    _filterPosition = -1;
1927                    //    return false; // Начинается иначе
1928                    //}
1929                    //_filterPosition++;
1930                    //if (_filterPosition == (_patternSequence.Length - 1))
1931                    //    return false;
1932                    //if (_filterPosition >= 0)
1933                    //{
1934                    //    if (element == _patternSequence[_filterPosition + 1])
1935                    //        _filterPosition++;
1936                    //    else
1937                    //        return false;
1938                    //}
1939                    //if (_filterPosition < 0)
1940                    //{
1941                    //    if (element == _patternSequence[0])
1942                    //        _filterPosition = 0;
1943                    //}
1944                }

1946            public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
1947            {
1948                foreach (var sequenceToMatch in sequencesToMatch)
1949                {
1950                    if (PatternMatch(sequenceToMatch))
1951                    {
1952                        _results.Add(sequenceToMatch);
1953                    }
1954                }
1955            }
1956        }

1958        #endregion
1959    }
1960 }
```

./Platform.Data.Doublets/Sequences/SequencesExtensions.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 SequencesExtensions
9       {
10          public static TLink Create<TLink>(this ILinks<TLink> sequences, IList<TLink[]>
          ↪ groupedSequence)
11          {
12              var finalSequence = new TLink[groupedSequence.Count];
13              for (var i = 0; i < finalSequence.Length; i++)
14              {
15                  var part = groupedSequence[i];
16                  finalSequence[i] = part.Length == 1 ? part[0] :
                  ↪ sequences.Create(part.ConvertToRestrictionsValues());
17              }
18              return sequences.Create(finalSequence.ConvertToRestrictionsValues());
```

```
19              }
20
21          public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
22          {
23              var list = new List<TLink>();
24              var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
25              sequences.Each(filler.AddAllValuesAndReturnConstant, new
   ↪   LinkAddress<TLink>(sequence));
26              return list;
27          }
28      }
29  }
```

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

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

```
                    }
                }
                var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
                if (UseCompression)
                {
                    if (LinksToSequenceConverter == null)
                    {
                        ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
                        if (UseSequenceMarker)
                        {
                            totalSequenceSymbolFrequencyCounter = new
                            ↪   TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                            ↪   MarkedSequenceMatcher);
                        }
                        else
                        {
                            totalSequenceSymbolFrequencyCounter = new
                            ↪   TotalSequenceSymbolFrequencyCounter<TLink>(links);
                        }
                        var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                        ↪   totalSequenceSymbolFrequencyCounter);
                        var compressingConverter = new CompressingConverter<TLink>(links,
                        ↪   balancedVariantConverter, doubletFrequenciesCache);
                        LinksToSequenceConverter = compressingConverter;
                    }
                }
                else
                {
                    if (LinksToSequenceConverter == null)
                    {
                        LinksToSequenceConverter = balancedVariantConverter;
                    }
                }
                if (UseIndex && Index == null)
                {
                    Index = new SequenceIndex<TLink>(links);
                }
                if (Walker == null)
                {
                    Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
                }
            }

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

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

```
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)]
```

```csharp
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

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

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

```csharp
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

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

```
71                     if (array.Length > 1)
72                     {
73                         ArrayPool.Free(array);
74                     }
75  #endif
76                     array = nextArray;
77                 }
78                 while (hasElements);
79                 var filledElementsCount = CountFilledElements(array);
80                 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)]
```

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

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

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

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

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

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

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

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

## ./Platform.Data.Doublets/SynchronizedLinks.cs

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

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

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

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

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

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

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

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

## ./Platform.Data.Doublets/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);
```

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

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

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)
        {
            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)
```

```
84                    {
85                        throw new ArgumentNullException(nameof(sb));
86                    }
87                    if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
    ↪   Constants.Itself)
88                    {
89                        return;
90                    }
91                    if (links.Exists(linkIndex))
92                    {
93                        if (visited.Add(linkIndex))
94                        {
95                            sb.Append('(');
96                            var link = new Link<ulong>(links.GetLink(linkIndex));
97                            if (renderIndex)
98                            {
99                                sb.Append(link.Index);
100                               sb.Append(':');
101                           }
102                           if (link.Source == link.Index)
103                           {
104                               sb.Append(link.Index);
105                           }
106                           else
107                           {
108                               var source = new Link<ulong>(links.GetLink(link.Source));
109                               if (isElement(source))
110                               {
111                                   appendElement(sb, source);
112                               }
113                               else
114                               {
115                                   links.AppendStructure(sb, visited, source.Index, isElement,
    ↪   appendElement, renderIndex);
116                               }
117                           }
118                           sb.Append(' ');
119                           if (link.Target == link.Index)
120                           {
121                               sb.Append(link.Index);
122                           }
123                           else
124                           {
125                               var target = new Link<ulong>(links.GetLink(link.Target));
126                               if (isElement(target))
127                               {
128                                   appendElement(sb, target);
129                               }
130                               else
131                               {
132                                   links.AppendStructure(sb, visited, target.Index, isElement,
    ↪   appendElement, renderIndex);
133                               }
134                           }
135                           sb.Append(')');
136                       }
137                       else
138                       {
139                           if (renderDebug)
140                           {
141                               sb.Append('*');
142                           }
143                           sb.Append(linkIndex);
144                       }
145                   }
146                   else
147                   {
148                       if (renderDebug)
149                       {
150                           sb.Append('~');
151                       }
152                       sb.Append(linkIndex);
153                   }
154               }
155           }
156   }
```

```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
        /// {
        ///     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;
```

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

```csharp
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;
193            }
194
195            public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
196            {
197                if (transaction.IsReverted)
198                {
199                    throw new InvalidOperationException("Transation is reverted.");
200                }
201                if (transaction.IsCommitted)
202                {
203                    throw new InvalidOperationException("Transation is commited.");
204                }
205            }
206
207            protected override void Dispose(bool manual, bool wasDisposed)
208            {
209                if (!wasDisposed && _layer != null && !_layer.IsDisposed)
210                {
211                    if (!IsCommitted && !IsReverted)
212                    {
213                        Revert();
214                    }
215                    _layer.ResetCurrentTransation();
216                }
217            }
218        }
219
220        public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
221
222        private readonly string _logAddress;
223        private readonly FileStream _log;
224        private readonly Queue<Transition> _transitions;
225        private readonly UniqueTimestampFactory _uniqueTimestampFactory;
226        private Task _transitionsPusher;
```

```csharp
227        private Transition _lastCommitedTransition;
228        private ulong _currentTransactionId;
229        private Queue<Transition> _currentTransactionTransitions;
230        private Transaction _currentTransaction;
231        private ulong _lastCommitedTransactionId;
232
233        public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
234            : base(links)
235        {
236            if (string.IsNullOrWhiteSpace(logAddress))
237            {
238                throw new ArgumentNullException(nameof(logAddress));
239            }
240            // В первой строке файла хранится последняя закоммиченную транзакцию.
241            // При запуске это используется для проверки удачного закрытия файла лога.
242            // In the first line of the file the last committed transaction is stored.
243            // On startup, this is used to check that the log file is successfully closed.
244            var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
245            var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
246            if (!lastCommitedTransition.Equals(lastWrittenTransition))
247            {
248                Dispose();
249                throw new NotSupportedException("Database is damaged, autorecovery is not
    ↪    supported yet.");
250            }
251            if (lastCommitedTransition.Equals(default(Transition)))
252            {
253                FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
254            }
255            _lastCommitedTransition = lastCommitedTransition;
256            // TODO: Think about a better way to calculate or store this value
257            var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
258            _lastCommitedTransactionId = allTransitions.Max(x => x.TransactionId);
259            _uniqueTimestampFactory = new UniqueTimestampFactory();
260            _logAddress = logAddress;
261            _log = FileHelpers.Append(logAddress);
262            _transitions = new Queue<Transition>();
263            _transitionsPusher = new Task(TransitionsPusher);
264            _transitionsPusher.Start();
265        }
266
267        public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
268
269        public override ulong Create(IList<ulong> restrictions)
270        {
271            var createdLinkIndex = Links.Create();
272            var createdLink = new Link<ulong>(Links.GetLink(createdLinkIndex));
273            CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪    default, createdLink));
274            return createdLinkIndex;
275        }
276
277        public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
278        {
279            var linkIndex = restrictions[Constants.IndexPart];
280            var beforeLink = new Link<ulong>(Links.GetLink(linkIndex));
281            linkIndex = Links.Update(restrictions, substitution);
282            var afterLink = new Link<ulong>(Links.GetLink(linkIndex));
283            CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪    beforeLink, afterLink));
284            return linkIndex;
285        }
286
287        public override void Delete(IList<ulong> restrictions)
288        {
289            var link = restrictions[Constants.IndexPart];
290            var deletedLink = new Link<ulong>(Links.GetLink(link));
291            Links.Delete(link);
292            CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪    deletedLink, default));
293        }
294
295        [MethodImpl(MethodImplOptions.AggressiveInlining)]
296        private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
    ↪    _transitions;
297
298        private void CommitTransition(Transition transition)
299        {
```

```csharp
            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;
            }
        }

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

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

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

```
378            }
379
380            #region DisposalBase
381
382            protected override void Dispose(bool manual, bool wasDisposed)
383            {
384                if (!wasDisposed)
385                {
386                    DisposeTransitions();
387                }
388                base.Dispose(manual, wasDisposed);
389            }
390
391            #endregion
392        }
393    }
```

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

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

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

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

```
31                    _index.Add(elements);
32                    var sequence = _listToSequenceLinkConverter.Convert(elements);
33                    return Links.GetOrCreate(sequence, _unicodeSequenceMarker);
34                }
35          }
36    }
```

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

```
 1    using System;
 2    using System.Collections.Generic;
 3    using System.Globalization;
 4    using System.Runtime.CompilerServices;
 5    using System.Text;
 6    using Platform.Data.Sequences;
 7
 8    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 9
10    namespace Platform.Data.Doublets.Unicode
11    {
12        public class UnicodeMap
13        {
14            public static readonly ulong FirstCharLink = 1;
15            public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
16            public static readonly ulong MapSize = 1 + char.MaxValue;
17
18            private readonly ILinks<ulong> _links;
19            private bool _initialized;
20
21            public UnicodeMap(ILinks<ulong> links) => _links = links;
22
23            public static UnicodeMap InitNew(ILinks<ulong> links)
24            {
25                var map = new UnicodeMap(links);
26                map.Init();
27                return map;
28            }
29
30            public void Init()
31            {
32                if (_initialized)
33                {
34                    return;
35                }
36                _initialized = true;
37                var firstLink = _links.CreatePoint();
38                if (firstLink != FirstCharLink)
39                {
40                    _links.Delete(firstLink);
41                }
42                else
43                {
44                    for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
45                    {
46                        // From NIL to It (NIL -> Character) transformation meaning, (or infinite
                        ↪   amount of NIL characters before actual Character)
47                        var createdLink = _links.CreatePoint();
48                        _links.Update(createdLink, firstLink, createdLink);
49                        if (createdLink != i)
50                        {
51                            throw new InvalidOperationException("Unable to initialize UTF 16
                            ↪   table.");
52                        }
53                    }
54                }
55            }
56
57            // 0 - null link
58            // 1 - nil character (0 character)
59            // ...
60            // 65536 (0(1) + 65535 = 65536 possible values)
61
62            [MethodImpl(MethodImplOptions.AggressiveInlining)]
63            public static ulong FromCharToLink(char character) => (ulong)character + 1;
64
65            [MethodImpl(MethodImplOptions.AggressiveInlining)]
66            public static char FromLinkToChar(ulong link) => (char)(link - 1);
67
68            [MethodImpl(MethodImplOptions.AggressiveInlining)]
69            public static bool IsCharLink(ulong link) => link <= MapSize;
70
```

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

        public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
        {
            var sb = new StringBuilder();
            if (links.Exists(link))
            {
                StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
                    x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                      element =>
                    {
                        sb.Append(FromLinkToChar(element));
                        return true;
                    });
            }
            return sb.ToString();
        }

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

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

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

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

```
147
148          public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
149          {
150               var result = new List<ulong[]>();
151               var offset = 0;
152               while (offset < array.Length)
153               {
154                    var relativeLength = 1;
155                    if (array[offset] <= LastCharLink)
156                    {
157                         var currentCategory =
                              ↪  CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
158                         var absoluteLength = offset + relativeLength;
159                         while (absoluteLength < array.Length &&
160                                array[absoluteLength] <= LastCharLink &&
161                                currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(⌋
                                   ↪  array[absoluteLength])))
162                         {
163                              relativeLength++;
164                              absoluteLength++;
165                         }
166                    }
167                    else
168                    {
169                         var absoluteLength = offset + relativeLength;
170                         while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
171                         {
172                              relativeLength++;
173                              absoluteLength++;
174                         }
175                    }
176                    // copy array
177                    var innerSequence = new ulong[relativeLength];
178                    var maxLength = offset + relativeLength;
179                    for (var i = offset; i < maxLength; i++)
180                    {
181                         innerSequence[i - offset] = array[i];
182                    }
183                    result.Add(innerSequence);
184                    offset += relativeLength;
185               }
186               return result;
187          }
188      }
189  }
```

./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs
```
1   using Platform.Interfaces;
2   using System.Collections.Generic;
3
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
```
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

```
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

```
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

```
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             });
44
45             Func<ulong, ulong, int> compareReference = comparer1.Compare;
46
47             var ts3 = Performance.Measure(() =>
48             {
49                 for (int i = 0; i < N; i++)
50                 {
51                     result = compareReference(x, y) >= 0;
52                 }
53             });
54
55             var comparer2 = new UInt64Comparer();
56
57             var ts4 = Performance.Measure(() =>
58             {
59                 for (int i = 0; i < N; i++)
60                 {
61                     result = comparer2.Compare(x, y) >= 0;
62                 }
63             });
64
65             Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");
66         }
67     }
68 }
```

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

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

```csharp
using Xunit;
using Platform.Diagnostics;

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

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

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

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

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

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

            ulong x = 10;
            ulong y = 500;

            bool result = false;

            var ts1 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Equals1(x, y);
                }
            });

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

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

            var equalityComparer1 = EqualityComparer<ulong>.Default;

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

            var equalityComparer2 = new UInt64EqualityComparer();

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

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

            var ts6 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
```

```csharp
                        result = equalityComparer3(x, y);
                    }
                });

                var comparer = Comparer<ulong>.Default;

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

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

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

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

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

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

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

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

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

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

```csharp
using Xunit;

namespace Platform.Data.Doublets.Tests
{
    public static class LinksConstantsTests
    {
        [Fact]
        public static void ExternalReferencesTest()
        {
            LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
                (long.MaxValue + 1UL, ulong.MaxValue));

            //var minimum = new Hybrid<ulong>(0, isExternal: true);
            var minimum = new Hybrid<ulong>(1, isExternal: true);
            var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);

            Assert.True(constants.IsExternalReference(minimum));
            Assert.True(constants.IsExternalReference(maximum));
        }
    }
}
```

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

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

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

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

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);

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

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

```csharp
49                var sequenceToItsLocalElementLevelsConverter = new
   ↪    SequenceToItsLocalElementLevelsConverter<ulong>(links,
   ↪    linkToItsFrequencyNumberConverter);
50                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
   ↪    sequenceToItsLocalElementLevelsConverter);
51
52                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
   ↪    Walker = new LeveledSequenceWalker<ulong>(links) });
53
54                ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
   ↪    index, optimalVariantConverter);
55            }
56        }
57
58        [Fact]
59        public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
60        {
61            using (var scope = new TempLinksTestScope(useSequences: false))
62            {
63                var links = scope.Links;
64
65                links.UseUnicode();
66
67                var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
68
69                var linksToFrequencies = new Dictionary<ulong, ulong>();
70
71                var totalSequenceSymbolFrequencyCounter = new
   ↪    TotalSequenceSymbolFrequencyCounter<ulong>(links);
72
73                var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
   ↪    totalSequenceSymbolFrequencyCounter);
74
75                var index = new
   ↪    CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
76                var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque↲
   ↪    ncyNumberConverter<ulong>(linkFrequenciesCache);
77
78                var sequenceToItsLocalElementLevelsConverter = new
   ↪    SequenceToItsLocalElementLevelsConverter<ulong>(links,
   ↪    linkToItsFrequencyNumberConverter);
79                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
   ↪    sequenceToItsLocalElementLevelsConverter);
80
81                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
   ↪    Walker = new LeveledSequenceWalker<ulong>(links) });
82
83                ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
   ↪    index, optimalVariantConverter);
84            }
85        }
86
87        private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
   ↪    SequenceToItsLocalElementLevelsConverter<ulong>
   ↪    sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
   ↪    OptimalVariantConverter<ulong> optimalVariantConverter)
88        {
89            index.Add(sequence);
90
91            var optimalVariant = optimalVariantConverter.Convert(sequence);
92
93            var readSequence1 = sequences.ToList(optimalVariant);
94
95            Assert.True(sequence.SequenceEqual(readSequence1));
96        }
97
98        [Fact]
99        public static void SavedSequencesOptimizationTest()
100       {
101           using (var memory = new HeapResizableDirectMemory())
102           using (var disposableLinks = new UInt64ResizableDirectMemoryLinks(memory))
103           {
104               var links = new UInt64Links(disposableLinks);
105
106               var meaningRoot = links.CreatePoint();
107               var unaryOne = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
108
109               var linksToFrequencies = new Dictionary<ulong, ulong>();
```

```
110        var totalSequenceSymbolFrequencyCounter = new
     ↪   TotalSequenceSymbolFrequencyCounter<ulong>(links);
111        var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
     ↪   totalSequenceSymbolFrequencyCounter);
112        var index = new
     ↪   CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
113        var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque⌟
     ↪   ncyNumberConverter<ulong>(linkFrequenciesCache);
114        var sequenceToItsLocalElementLevelsConverter = new
     ↪   SequenceToItsLocalElementLevelsConverter<ulong>(links,
     ↪   linkToItsFrequencyNumberConverter);
115        var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
     ↪   sequenceToItsLocalElementLevelsConverter);
116
117        var sequencesOptions = new SequencesOptions<ulong>()
118        {
119
120        };
121
122        var sequences = new Sequences.Sequences(new SynchronizedLinks<ulong>(links));
123
124
125    }
126    // create some sequences
127    // get list of sequences links
128    // for each sequence link
129    //   create new sequence version
130    //   if new sequence is not the same as sequence link
131    //     delete sequence link
132    //     collect garbadge
133    }
134    }
135 }
```

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

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

```
45                                                  readSequence2.Add);
46                      sw3.Stop();
47
48                      Assert.True(sequence.SequenceEqual(readSequence1));
49
50                      Assert.True(sequence.SequenceEqual(readSequence2));
51
52                      // Assert.True(sw2.Elapsed < sw3.Elapsed);
53
54                      Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
                      ↪  {sw2.Elapsed}");
55
56                      for (var i = 0; i < sequenceLength; i++)
57                      {
58                          links.Delete(sequence[i]);
59                      }
60                  }
61              }
62          }
63  }
```

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

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

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

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

```
1  using Xunit;
2  using Platform.Scopes;
3  using Platform.Memory;
4  using Platform.Data.Doublets.Decorators;
5  using Platform.Reflection;
6  using Platform.Data.Doublets.ResizableDirectMemory.Generic;
7  using Platform.Data.Doublets.ResizableDirectMemory.Specific;
8
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         }
23
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;
```

```csharp
using Platform.Collections;
using Platform.Random;
using Platform.IO;
using Platform.Singletons;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Unicode;

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

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

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

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                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 sw1 = Stopwatch.StartNew();
                var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();

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

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

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

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

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

        //     const long sequenceLength = 8;

        //     const ulong itself = LinksConstants.Itself;

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

        //         SequencesOptions o = new SequencesOptions();

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

        //         var sequences = new Sequences(links);
```

```csharp
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
128                 var sw2 = Stopwatch.StartNew();
129                 var searchResults2 = sequences.Each1(sequence); sw2.Stop();
130
131                 var sw3 = Stopwatch.StartNew();
132                 var searchResults3 = sequences.Each(sequence.ConvertToRestrictionsValues());
                 ↪   sw3.Stop();
133
134                 var intersection0 = createResults.Intersect(searchResults0).ToList();
135                 Assert.True(intersection0.Count == searchResults0.Count);
136                 Assert.True(intersection0.Count == createResults.Length);
137
138                 var intersection1 = createResults.Intersect(searchResults1).ToList();
139                 Assert.True(intersection1.Count == searchResults1.Count);
140                 Assert.True(intersection1.Count == createResults.Length);
141
142                 var intersection2 = createResults.Intersect(searchResults2).ToList();
143                 Assert.True(intersection2.Count == searchResults2.Count);
144                 Assert.True(intersection2.Count == createResults.Length);
145
146                 var intersection3 = createResults.Intersect(searchResults3).ToList();
147                 Assert.True(intersection3.Count == searchResults3.Count);
148                 Assert.True(intersection3.Count == createResults.Length);
149
150                 for (var i = 0; i < sequenceLength; i++)
151                 {
152                     links.Delete(sequence[i]);
153                 }
154             }
155         }
156
157         [Fact]
158         public static void BalancedVariantSearchTest()
159         {
160             const long sequenceLength = 200;
161
162             using (var scope = new TempLinksTestScope(useSequences: true))
```

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

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

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

                        var 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;

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

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

                        var createResults = sequences.CreateAllVariants2(sequence);

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

                        var partialSequence = new ulong[sequenceLength - 2];

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

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

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

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

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

```csharp
237
238             //Global.Trash = searchResults3;
239
240             //var searchResults1Strings = searchResults1.Select(x => x + ": " +
    ↪   sequences.FormatSequence(x)).ToList();
241             //Global.Trash = searchResults1Strings;
242
243             var intersection1 = createResults.Intersect(searchResults1).ToList();
244             Assert.True(intersection1.Count == createResults.Length);
245
246             var intersection2 = createResults.Intersect(searchResults2).ToList();
247             Assert.True(intersection2.Count == createResults.Length);
248
249             var intersection4 = createResults.Intersect(searchResults4).ToList();
250             Assert.True(intersection4.Count == createResults.Length);
251
252             for (var i = 0; i < sequenceLength; i++)
253             {
254                 links.Delete(sequence[i]);
255             }
256         }
257     }
258
259     [Fact]
260     public static void BalancedPartialVariantsSearchTest()
261     {
262         const long sequenceLength = 200;
263
264         using (var scope = new TempLinksTestScope(useSequences: true))
265         {
266             var links = scope.Links;
267             var sequences = scope.Sequences;
268
269             var sequence = new ulong[sequenceLength];
270             for (var i = 0; i < sequenceLength; i++)
271             {
272                 sequence[i] = links.Create();
273             }
274
275             var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
276
277             var balancedVariant = balancedVariantConverter.Convert(sequence);
278
279             var partialSequence = new ulong[sequenceLength - 2];
280
281             Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
282
283             var sw1 = Stopwatch.StartNew();
284             var searchResults1 =
    ↪   sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
285
286             var sw2 = Stopwatch.StartNew();
287             var searchResults2 =
    ↪   sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
288
289             Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
290
291             Assert.True(searchResults2.Count == 1 && balancedVariant ==
    ↪   searchResults2.First());
292
293             for (var i = 0; i < sequenceLength; i++)
294             {
295                 links.Delete(sequence[i]);
296             }
297         }
298     }
299
300     [Fact(Skip = "Correct implementation is pending")]
301     public static void PatternMatchTest()
302     {
303         var zeroOrMany = Sequences.Sequences.ZeroOrMany;
304
305         using (var scope = new TempLinksTestScope(useSequences: true))
306         {
307             var links = scope.Links;
308             var sequences = scope.Sequences;
309
310             var e1 = links.Create();
311             var e2 = links.Create();
312
```

```csharp
313                    var sequence = new[]
314                    {
315                        e1, e2, e1, e2 // mama / papa
316                    };
317
318                    var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
319
320                    var balancedVariant = balancedVariantConverter.Convert(sequence);
321
322                    // 1: [1]
323                    // 2: [2]
324                    // 3: [1,2]
325                    // 4: [1,2,1,2]
326
327                    var doublet = links.GetSource(balancedVariant);
328
329                    var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
330
331                    Assert.True(matchedSequences1.Count == 0);
332
333                    var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
334
335                    Assert.True(matchedSequences2.Count == 0);
336
337                    var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
338
339                    Assert.True(matchedSequences3.Count == 0);
340
341                    var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
342
343                    Assert.Contains(doublet, matchedSequences4);
344                    Assert.Contains(balancedVariant, matchedSequences4);
345
346                    for (var i = 0; i < sequence.Length; i++)
347                    {
348                        links.Delete(sequence[i]);
349                    }
350                }
351            }
352
353            [Fact]
354            public static void IndexTest()
355            {
356                using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
                     true }, useSequences: true))
357                {
358                    var links = scope.Links;
359                    var sequences = scope.Sequences;
360                    var index = sequences.Options.Index;
361
362                    var e1 = links.Create();
363                    var e2 = links.Create();
364
365                    var sequence = new[]
366                    {
367                        e1, e2, e1, e2 // mama / papa
368                    };
369
370                    Assert.False(index.MightContain(sequence));
371
372                    index.Add(sequence);
373
374                    Assert.True(index.MightContain(sequence));
375                }
376            }
377
378            /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
                 D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
                 %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
379            private static readonly string _exampleText =
380                @"([english
                 version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
381
382 Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов
    (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там
    где есть место для нового начала? Разве пустота это не характеристика пространства?
    Пространство это то, что можно чем-то наполнить?
383
```

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

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

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

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

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

                var compressedVariant = compressingConverter.Convert(sequence);

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

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

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

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

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

            [Fact]
            public static void CompressionEfficiencyTest()
            {
                var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
                    StringSplitOptions.RemoveEmptyEntries);
                var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
                var totalCharacters = arrays.Select(x => x.Length).Sum();

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

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

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

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

                    var sequences = compressor3;
                    //var meaningRoot = links.CreatePoint();
                    //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
                    //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
                    //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                        constants.Itself);

                    //var unaryNumberToAddressConverter = new
                        UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
                    //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
                        unaryOne);
                    //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                        frequencyMarker, unaryOne, unaryNumberIncrementer);
                    //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
                        frequencyPropertyMarker, frequencyMarker);
                    //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                        frequencyPropertyOperator, frequencyIncrementer);
                    //var linkToItsFrequencyNumberConverter = new
                        LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                        unaryNumberToAddressConverter);
```

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

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

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

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

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

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

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

544             var sw1 = Stopwatch.StartNew();

546             for (int i = START; i < END; i++)
547             {
548                 linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
549                 compressed1[i] = compressor1.Convert(arrays[i]);
550             }

552             var elapsed1 = sw1.Elapsed;

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

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

558             var sw2 = Stopwatch.StartNew();

560             for (int i = START; i < END; i++)
561             {
562                 compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
563             }

565             var elapsed2 = sw2.Elapsed;

567             for (int i = START; i < END; i++)
568             {
569                 linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
570             }

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

574             var sw3 = Stopwatch.StartNew();

576             for (int i = START; i < END; i++)
577             {
578                 //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
579                 compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
580             }

582             var elapsed3 = sw3.Elapsed;

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

586             // Assert.True(elapsed1 > elapsed2);

588             // Checks
589             for (int i = START; i < END; i++)
590             {
591                 var sequence1 = compressed1[i];
592                 var sequence2 = compressed2[i];
593                 var sequence3 = compressed3[i];

595                 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);

                    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;
```

```csharp
            const ulong maxNumbers = 12500;

            var strings = new List<string>();

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

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

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

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

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

                var sw1 = Stopwatch.StartNew();

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

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

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

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

```csharp
                    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();
                }
        }

        [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();
```

```
801             scope2.Links.UseUnicode();
802
803                 var compressor1 = scope1.Sequences;
804                 var compressor2 = scope2.Sequences;
805
806                 var compressed1 = new ulong[arrays.Length];
807                 var compressed2 = new ulong[arrays.Length];
808
809                 var sw1 = Stopwatch.StartNew();
810
811                 var START = 0;
812                 var END = arrays.Length;
813
814                 for (int i = START; i < END; i++)
815                 {
816                     compressed1[i] = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
817                 }
818
819                 var elapsed1 = sw1.Elapsed;
820
821                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
822
823                 var sw2 = Stopwatch.StartNew();
824
825                 for (int i = START; i < END; i++)
826                 {
827                     compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
828                 }
829
830                 var elapsed2 = sw2.Elapsed;
831
832                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
    ↪    {elapsed2}");
833
834                 Assert.True(elapsed1 > elapsed2);
835
836                 // Checks
837                 for (int i = START; i < END; i++)
838                 {
839                     var sequence1 = compressed1[i];
840                     var sequence2 = compressed2[i];
841
842                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
843                     {
844                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
    ↪        scope1.Links);
845                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
    ↪        scope2.Links);
846
847                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
848                     }
849                 }
850
851
852                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
853                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
854
855                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
    ↪    totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
    ↪    totalCharacters}");
856
857                 // Can be worse than balanced variant
858                 //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
859
860                 //compressor1.ValidateFrequencies();
861             }
862         }
863
864         [Fact]
865         public static void AllTreeBreakDownAtSequencesCreationBugTest()
866         {
867             // Made out of AllPossibleConnectionsTest test.
868
869             //const long sequenceLength = 5; //100% bug
870             const long sequenceLength = 4; //100% bug
871             //const long sequenceLength = 3; //100% _no_bug (ok)
872
873             using (var scope = new TempLinksTestScope(useSequences: true))
874             {
875                 var links = scope.Links;
```

```csharp
                    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();

                        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()
```

```
956                 {
957                     const long sequenceLength = 3;
958
959                     using (var scope = new TempLinksTestScope(useSequences: true))
960                     {
961                         var links = scope.Links;
962                         var sequences = scope.Sequences;
963
964                         var sequence = new ulong[sequenceLength];
965                         for (var i = 0; i < sequenceLength; i++)
966                         {
967                             sequence[i] = links.Create();
968                         }
969
970                         var createResults = sequences.CreateAllVariants2(sequence);
971
972                         //var reverseResults =
                      ↪   sequences.CreateAllVariants2(sequence.Reverse().ToArray());
973
974                         for (var i = 0; i < 1; i++)
975                         {
976                             var linksTotalUsages1 = new ulong[links.Count() + 1];
977
978                             sequences.CalculateAllUsages(linksTotalUsages1);
979
980                             var linksTotalUsages2 = new ulong[links.Count() + 1];
981
982                             sequences.CalculateAllUsages2(linksTotalUsages2);
983
984                             var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
985                             Assert.True(intersection1.Count == linksTotalUsages2.Length);
986                         }
987
988                         for (var i = 0; i < sequenceLength; i++)
989                         {
990                             links.Delete(sequence[i]);
991                         }
992                     }
993                 }
994             }
995     }
```

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

```
1   using System.IO;
2   using Platform.Disposables;
3   using Platform.Data.Doublets.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  }
```

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

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

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

        private const long Iterations = 10 * 1024;

        #region Concept

        [Fact]
        public static void MultipleCreateAndDeleteTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
                UInt64ResizableDirectMemoryLinks>>())
            {
                new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeleti
                    ons(100);
            }
        }

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

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

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

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

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

                l2 = links.Update(l2, l1);

                links.Delete(l2);

                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;
```

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

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

                            l2 = links.Update(l2, l1);

                            links.Delete(l2);

                            ExceptionThrower();

                            transaction.Commit();
                        }

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

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

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

                    lastScope.DeleteFiles();
                }
            }

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

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

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

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

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

                    links.Unsync.DisposeIfPossible();

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

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

                        links.Delete(l2);

                        ExceptionThrower();

                        transaction.Commit();
                    }

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

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

                lastScope.DeleteFiles();
            }
        }

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

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

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

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

                    links.Delete(l1);

                    transaction.Commit();
                }
```

```
244                    Global.Trash = links.Count();
245                }
246
247            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran ↵
                    ↪  sactionLogFilename);
248            }
249
250            [Fact]
251            public static void TransactionDamage()
252            {
253                var itself = _constants.Itself;
254
255                var tempDatabaseFilename = Path.GetTempFileName();
256                var tempTransactionLogFilename = Path.GetTempFileName();
257
258                // Commit
259                using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                    ↪  tempTransactionLogFilename))
260                using (var links = new UInt64Links(memoryAdapter))
261                {
262                    using (var transaction = memoryAdapter.BeginTransaction())
263                    {
264                        var l1 = links.CreateAndUpdate(itself, itself);
265                        var l2 = links.CreateAndUpdate(itself, itself);
266
267                        Global.Trash = links.Update(l2, l2, l1, l2);
268
269                        links.Delete(l1);
270
271                        transaction.Commit();
272                    }
273
274                    Global.Trash = links.Count();
275                }
276
277                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran ↵
                    ↪  sactionLogFilename);
278
279                // Damage database
280
281                FileHelpers.WriteFirst(tempTransactionLogFilename, new
                    ↪  UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
282
283                // Try load damaged database
284                try
285                {
286                    // TODO: Fix
287                    using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                        ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                        ↪  tempTransactionLogFilename))
288                    using (var links = new UInt64Links(memoryAdapter))
289                    {
290                        Global.Trash = links.Count();
291                    }
292                }
293                catch (NotSupportedException ex)
294                {
295                    Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
                        ↪  yet.");
296                }
297
298                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran ↵
                    ↪  sactionLogFilename);
299
300                File.Delete(tempDatabaseFilename);
301                File.Delete(tempTransactionLogFilename);
302            }
303
304            [Fact]
305            public static void Bug1Test()
306            {
307                var tempDatabaseFilename = Path.GetTempFileName();
308                var tempTransactionLogFilename = Path.GetTempFileName();
309
310                var itself = _constants.Itself;
311
312                // User Code Error (Autoreverted), some data saved
313                try
```

```csharp
            {
                ulong l1;
                ulong l2;

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

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

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

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

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

                        links.Delete(l2);

                        ExceptionThrower();

                        transaction.Commit();
                    }

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

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

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

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

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

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

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

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

```
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();
```

```csharp
                //links.TestParallelForeach();
            }

            links.TestDeletionOfAllLinks();

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

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

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

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

        GetBaseRandomLoopOverhead(linksStep);
        GetBaseRandomLoopOverhead(linksStep);

        var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);

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

        var loops = totalLinksToCreate / linksStep;

        for (int i = 0; i < loops; i++)
        {
            creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
            searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));

            Console.Write("\rC + S {0}/{1}", i + 1, loops);
        }

        ConsoleHelpers.Debug();

        for (int i = 0; i < loops; i++)
        {
            deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));

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

        ConsoleHelpers.Debug();

        ConsoleHelpers.Debug("C S D");

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

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

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

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

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

```csharp
                    links.Create(0, 0);
            }

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

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

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

                ulong counter = 0;

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

                var sw = Stopwatch.StartNew();

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

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

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

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

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

                long counter = 0;

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

                var sw = Stopwatch.StartNew();

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

```
607                    });
608
609                    var elapsedTime = sw.Elapsed;
610
611                    var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
612
613                    links.Delete(firstLink);
614
615                    ConsoleHelpers.Debug(
616                        "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                           ↪ second), counter result: {3}",
617                        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
618                }
619            }
620
621            [Fact(Skip = "performance test")]
622            public static void TestGetTarget()
623            {
624                using (var scope = new TempLinksTestScope())
625                {
626                    var links = scope.Links;
627                    ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
                       ↪ Iterations);
628
629                    ulong counter = 0;
630
631                    //var firstLink = links.First();
632                    var firstLink = links.Create();
633
634                    var sw = Stopwatch.StartNew();
635
636                    for (ulong i = 0; i < Iterations; i++)
637                    {
638                        counter += links.GetTarget(firstLink);
639                    }
640
641                    var elapsedTime = sw.Elapsed;
642
643                    var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
644
645                    links.Delete(firstLink);
646
647                    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                }
718
719                File.Delete(tempFilename);
720            }*/
721
722            [Fact(Skip = "useless: O(0), was dependent on creation tests")]
723            public static void TestRandomSearchAll()
724            {
725                using (var scope = new TempLinksTestScope())
726                {
727                    var links = scope.Links;
728                    ulong counter = 0;
729
730                    var maxLink = links.Count();
731
732                    var iterations = links.Count();
733
734                    ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
                       ↪ links.Count());
735
736                    var sw = Stopwatch.StartNew();
737
738                    for (var i = iterations; i > 0; i--)
739                    {
740                        var linksAddressRange = new
                           ↪ Range<ulong>(_constants.PossibleInnerReferencesRange.Minimum, maxLink);
741
742                        var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
743                        var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
744
745                        counter += links.SearchOrDefault(source, target);
746                    }
747
748                    var elapsedTime = sw.Elapsed;
749
750                    var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
751
752                    ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
                       ↪ Iterations per second), c: {3}",
```

```csharp
753                            iterations, elapsedTime, (long)iterationsPerSecond, counter);
754                    }
755            }
756
757            [Fact(Skip = "useless: O(0), was dependent on creation tests")]
758            public static void TestEach()
759            {
760                using (var scope = new TempLinksTestScope())
761                {
762                    var links = scope.Links;
763
764                    var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
765
766                    ConsoleHelpers.Debug("Testing Each function.");
767
768                    var sw = Stopwatch.StartNew();
769
770                    links.Each(counter.IncrementAndReturnTrue);
771
772                    var elapsedTime = sw.Elapsed;
773
774                    var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
775
776                    ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
           ↪  links per second)",
777                        counter, elapsedTime, (long)linksPerSecond);
778                }
779            }
780
781            /*
782            [Fact]
783            public static void TestForeach()
784            {
785                var tempFilename = Path.GetTempFileName();
786
787                using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
       ↪  DefaultLinksSizeStep))
788                {
789                    ulong counter = 0;
790
791                    ConsoleHelpers.Debug("Testing foreach through links.");
792
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

```
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;
```

```csharp
using Platform.Data.Doublets.Unicode;
using Platform.Data.Doublets.ResizableDirectMemory.Generic;

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

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

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

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

                var itself = links.Constants.Itself;

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

                var powerOf2ToUnaryNumberConverter = new
                    PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
```

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

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

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

                        var originalString = "Hello";

                        var unicodeSequenceLink =
                        ↪  stringToUnicodeSequenceConverter.Convert(originalString);

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

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

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

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

                        var resultingString =
                        ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);

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

# Index