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

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

using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
    {
        private ILinks<TLink> _facade;

        public LinksConstants<TLink> Constants { get; }

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

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    public abstract class LinksDisposableDecoratorBase<TLink> : DisposableBase, ILinks<TLink>
    {
        private ILinks<TLink> _facade;

        public LinksConstants<TLink> Constants { get; }

        public ILinks<TLink> Links { get; }

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

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

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

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

```

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

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

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

        protected override bool AllowMultipleDisposeCalls => true;

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
            Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
            Links.EnsureInnerReferenceExists(substitution, nameof(substitution));
            return Links.Update(restrictions, substitution);
        }

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

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

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

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

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

```

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            var constants = Constants;
            Links.EnsureCreated(substitution[constants.SourcePart],
             ↪  substitution[constants.TargetPart]);
            return Links.Update(restrictions, substitution);
        }
    }
}
```

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

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

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

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

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

```
        {
            var link = Links.Create();
            return Links.Update(link, link, link);
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// Представляет объект для работы с базой данных (файлом) в формате Links (массива связей).
    /// </summary>
    /// <remarks>
    /// Возможные оптимизации:
    /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
    ///     + меньше объём БД
    ///     - меньше производительность
    ///     - больше ограничение на количество связей в БД)
    /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
    ///     + меньше объём БД
    ///     - больше сложность
    ///
    /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
    ↪  поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
    ↪  460 752 303 423 488
    /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
    ↪  (битовыми строками) - вариант матрицы (выстраеваемой лениво).
```

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

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

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

```
18      /// </remarks>
19      internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
20      {
21          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
22
23          public UniLinks(ILinks<TLink> links) : base(links) { }
24
25          private struct Transition
26          {
27              public IList<TLink> Before;
28              public IList<TLink> After;
29
30              public Transition(IList<TLink> before, IList<TLink> after)
31              {
32                  Before = before;
33                  After = after;
34              }
35          }
36
37          //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
38          //public static readonly IReadOnlyList<TLink> NullLink = new
            ↪ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
            ↪ });
39
40          // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
            ↪ (Links-Expression)
41          public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
            ↪ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            ↪ substitutedHandler)
42          {
43              ////List<Transition> transitions = null;
44              ////if (!restriction.IsNullOrEmpty())
45              ////{
46              ////    // Есть причина делать проход (чтение)
47              ////    if (matchedHandler != null)
48              ////    {
49              ////        if (!substitution.IsNullOrEmpty())
50              ////        {
51              ////            // restriction => { 0, 0, 0 } | { 0 } // Create
52              ////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
                ↪ Create / Update
53              ////            // substitution => { 0, 0, 0 } | { 0 } // Delete
54              ////            transitions = new List<Transition>();
55              ////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
56              ////            {
57              ////                // If index is Null, that means we always ignore every other
                ↪ value (they are also Null by definition)
58              ////                var matchDecision = matchedHandler(, NullLink);
59              ////                if (Equals(matchDecision, Constants.Break))
60              ////                    return false;
61              ////                if (!Equals(matchDecision, Constants.Skip))
62              ////                    transitions.Add(new Transition(matchedLink, newValue));
63              ////            }
64              ////            else
65              ////            {
66              ////                Func<T, bool> handler;
67              ////                handler = link =>
68              ////                {
69              ////                    var matchedLink = Memory.GetLinkValue(link);
70              ////                    var newValue = Memory.GetLinkValue(link);
71              ////                    newValue[Constants.IndexPart] = Constants.Itself;
72              ////                    newValue[Constants.SourcePart] =
                ↪ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
                ↪ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
73              ////                    newValue[Constants.TargetPart] =
                ↪ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
                ↪ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
74              ////                    var matchDecision = matchedHandler(matchedLink, newValue);
75              ////                    if (Equals(matchDecision, Constants.Break))
76              ////                        return false;
77              ////                    if (!Equals(matchDecision, Constants.Skip))
78              ////                        transitions.Add(new Transition(matchedLink, newValue));
79              ////                    return true;
80              ////                };
81              ////                if (!Memory.Each(handler, restriction))
82              ////                    return Constants.Break;
83              ////            }
```

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

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

```
161             //        };
162             //        if (!Memory.Each(handler, restriction))
163             //            return Constants.Break;
164             //    }
165             //    if (!matchedLinks.IsNullOrEmpty())
166             //    {
167             //        var totalMatchedLinks = matchedLinks.Count;
168             //        for (var i = 0; i < totalMatchedLinks; i++)
169             //        {
170             //            var matchedLink = matchedLinks[i];
171             //            if (substitutedHandler != null)
172             //            {
173             //                var newValue = new List<T>(); // TODO: Prepare value to update here
174             //                // TODO: Decide is it actually needed to use Before and After
                ↪  substitution handling.
175             //                var substitutedDecision = substitutedHandler(matchedLink,
                ↪  newValue);
176             //                if (Equals(substitutedDecision, Constants.Break))
177             //                    return Constants.Break;
178             //                if (Equals(substitutedDecision, Constants.Continue))
179             //                {
180             //                    // Actual update here
181             //                    Memory.SetLinkValue(newValue);
182             //                }
183             //                if (Equals(substitutedDecision, Constants.Skip))
184             //                {
185             //                    // Cancel the update. TODO: decide use separate Cancel
                ↪  constant or Skip is enough?
186             //                }
187             //            }
188             //        }
189             //    }
190             //}
191             return Constants.Continue;
192         }
193
194         public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
            ↪  matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            ↪  substitutionHandler)
195         {
196             if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
197             {
198                 return Constants.Continue;
199             }
200             else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
                ↪  Check if it is a correct condition
201             {
202                 // Or it only applies to trigger without matchHandler.
203                 throw new NotImplementedException();
204             }
205             else if (!substitution.IsNullOrEmpty()) // Creation
206             {
207                 var before = ArrayPool<TLink>.Empty;
208                 // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
                    ↪  (пройти мимо) или пустить (взять)?
209                 if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  Constants.Break))
210                 {
211                     return Constants.Break;
212                 }
213                 var after = (IList<TLink>)substitution.ToArray();
214                 if (_equalityComparer.Equals(after[0], default))
215                 {
216                     var newLink = Links.Create();
217                     after[0] = newLink;
218                 }
219                 if (substitution.Count == 1)
220                 {
221                     after = Links.GetLink(substitution[0]);
222                 }
223                 else if (substitution.Count == 3)
224                 {
225                     //Links.Create(after);
226                 }
227                 else
228                 {
229                     throw new NotSupportedException();
230                 }
```

```
231                 if (matchHandler != null)
232                 {
233                     return substitutionHandler(before, after);
234                 }
235                 return Constants.Continue;
236             }
237             else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
238             {
239                 if (patternOrCondition.Count == 1)
240                 {
241                     var linkToDelete = patternOrCondition[0];
242                     var before = Links.GetLink(linkToDelete);
243                     if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                        ↪ Constants.Break))
244                     {
245                         return Constants.Break;
246                     }
247                     var after = ArrayPool<TLink>.Empty;
248                     Links.Update(linkToDelete, Constants.Null, Constants.Null);
249                     Links.Delete(linkToDelete);
250                     if (matchHandler != null)
251                     {
252                         return substitutionHandler(before, after);
253                     }
254                     return Constants.Continue;
255                 }
256                 else
257                 {
258                     throw new NotSupportedException();
259                 }
260             }
261             else // Replace / Update
262             {
263                 if (patternOrCondition.Count == 1) //-V3125
264                 {
265                     var linkToUpdate = patternOrCondition[0];
266                     var before = Links.GetLink(linkToUpdate);
267                     if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                        ↪ Constants.Break))
268                     {
269                         return Constants.Break;
270                     }
271                     var after = (IList<TLink>)substitution.ToArray(); //-V3125
272                     if (_equalityComparer.Equals(after[0], default))
273                     {
274                         after[0] = linkToUpdate;
275                     }
276                     if (substitution.Count == 1)
277                     {
278                         if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
279                         {
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>
```

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

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

./Platform.Data.Doublets/DoubletComparer.cs

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

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

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

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

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

./Platform.Data.Doublets/Doublet.cs

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

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

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

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

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

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

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

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

```

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

./Platform.Data.Doublets/Hybrid.cs

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

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

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

        static Hybrid()
        {
            _absAndConvert = DelegateHelpers.Compile<Func<object, T>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<T>();
                emiter.LoadArgument(0);
                var signedVersion = NumericType<T>.SignedVersion;
                var signedVersionField =
                ↪  typeof(NumericType<T>).GetTypeInfo().GetField("SignedVersion",
                ↪  BindingFlags.Static | BindingFlags.Public);
                //emiter.LoadField(signedVersionField);
                emiter.Emit(OpCodes.Ldsfld, signedVersionField);
                var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
                ↪  Types<object, Type>.Array);
                emiter.Call(changeTypeMethod);
                emiter.UnboxValue(signedVersion);
                var absMethod = typeof(Math).GetTypeInfo().GetMethod("Abs", new[] {
                ↪  signedVersion });
                emiter.Call(absMethod);
                var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                ↪  signedVersion });
                emiter.Call(unsignedMethod);
                emiter.Return();
            });
            _absAndNegateAndConvert = DelegateHelpers.Compile<Func<object, T>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<T>();
                emiter.LoadArgument(0);
                var signedVersion = NumericType<T>.SignedVersion;
                var signedVersionField =
                ↪  typeof(NumericType<T>).GetTypeInfo().GetField("SignedVersion",
                ↪  BindingFlags.Static | BindingFlags.Public);
                //emiter.LoadField(signedVersionField);
                emiter.Emit(OpCodes.Ldsfld, signedVersionField);
                var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
                ↪  Types<object, Type>.Array);
                emiter.Call(changeTypeMethod);
                emiter.UnboxValue(signedVersion);
                var absMethod = typeof(Math).GetTypeInfo().GetMethod("Abs", new[] {
                ↪  signedVersion });
                emiter.Call(absMethod);
                var negateMethod = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate
                ↪  ").MakeGenericMethod(signedVersion);
                emiter.Call(negateMethod);
                var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                ↪  signedVersion });
                emiter.Call(unsignedMethod);
                emiter.Return();
            });
        }

        public readonly T Value;
        public bool IsNothing => Convert.ToInt64(To.Signed(Value)) == 0;
        public bool IsInternal => Convert.ToInt64(To.Signed(Value)) > 0;
        public bool IsExternal => Convert.ToInt64(To.Signed(Value)) < 0;
        public long AbsoluteValue =>
        ↪  Platform.Numbers.Math.Abs(Convert.ToInt64(To.Signed(Value)));
```

```
62
63          public Hybrid(T value)
64          {
65              Ensure.OnDebug.IsUnsignedInteger<T>();
66              Value = value;
67          }
68
69          public Hybrid(object value) => Value = To.UnsignedAs<T>(Convert.ChangeType(value,
            ↪  NumericType<T>.SignedVersion));
70
71          public Hybrid(object value, bool isExternal)
72          {
73              //var signedType = Type<T>.SignedVersion;
74              //var signedValue = Convert.ChangeType(value, signedType);
75              //var abs = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Abs").MakeGeneric
                ↪  Method(signedType);
76              //var negate = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate").MakeG
                ↪  enericMethod(signedType);
77              //var absoluteValue = abs.Invoke(null, new[] { signedValue });
78              //var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
                ↪  absoluteValue;
79              //Value = To.UnsignedAs<T>(resultValue);
80              if (isExternal)
81              {
82                  Value = _absAndNegateAndConvert(value);
83              }
84              else
85              {
86                  Value = _absAndConvert(value);
87              }
88          }
89
90          public static implicit operator Hybrid<T>(T integer) => new Hybrid<T>(integer);
91
92          public static explicit operator Hybrid<T>(ulong integer) => new Hybrid<T>(integer);
93
94          public static explicit operator Hybrid<T>(long integer) => new Hybrid<T>(integer);
95
96          public static explicit operator Hybrid<T>(uint integer) => new Hybrid<T>(integer);
97
98          public static explicit operator Hybrid<T>(int integer) => new Hybrid<T>(integer);
99
100         public static explicit operator Hybrid<T>(ushort integer) => new Hybrid<T>(integer);
101
102         public static explicit operator Hybrid<T>(short integer) => new Hybrid<T>(integer);
103
104         public static explicit operator Hybrid<T>(byte integer) => new Hybrid<T>(integer);
105
106         public static explicit operator Hybrid<T>(sbyte integer) => new Hybrid<T>(integer);
107
108         public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;
109
110         public static explicit operator ulong(Hybrid<T> hybrid) =>
            ↪  Convert.ToUInt64(hybrid.Value);
111
112         public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;
113
114         public static explicit operator uint(Hybrid<T> hybrid) => Convert.ToUInt32(hybrid.Value);
115
116         public static explicit operator int(Hybrid<T> hybrid) =>
            ↪  Convert.ToInt32(hybrid.AbsoluteValue);
117
118         public static explicit operator ushort(Hybrid<T> hybrid) =>
            ↪  Convert.ToUInt16(hybrid.Value);
119
120         public static explicit operator short(Hybrid<T> hybrid) =>
            ↪  Convert.ToInt16(hybrid.AbsoluteValue);
121
122         public static explicit operator byte(Hybrid<T> hybrid) => Convert.ToByte(hybrid.Value);
123
124         public static explicit operator sbyte(Hybrid<T> hybrid) =>
            ↪  Convert.ToSByte(hybrid.AbsoluteValue);
125
126         public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
            ↪  default(T).ToString() : IsExternal ? $"<{AbsoluteValue}>" : Value.ToString();
127     }
128 }
```

./Platform.Data.Doublets/ILinks.cs

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

using System.Collections.Generic;

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

./Platform.Data.Doublets/ILinksExtensions.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using Platform.Ranges;
using Platform.Collections.Arrays;
using Platform.Numbers;
using Platform.Random;
using Platform.Setters;
using Platform.Data.Exceptions;
using Platform.Data.Doublets.Decorators;

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

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

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

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

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

        /// <remarks>
        /// TODO: Возможно есть очень простой способ это сделать.
        /// (Например просто удалить файл, или изменить его размер таким образом,
```

```
62          /// чтобы удалился весь контент)
63          /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
64          /// </remarks>
65          public static void DeleteAll<TLink>(this ILinks<TLink> links)
66          {
67              var equalityComparer = EqualityComparer<TLink>.Default;
68              var comparer = Comparer<TLink>.Default;
69              for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
                ↪ Arithmetic.Decrement(i))
70              {
71                  links.Delete(i);
72                  if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
73                  {
74                      i = links.Count();
75                  }
76              }
77          }
78
79          public static TLink First<TLink>(this ILinks<TLink> links)
80          {
81              TLink firstLink = default;
82              var equalityComparer = EqualityComparer<TLink>.Default;
83              if (equalityComparer.Equals(links.Count(), default))
84              {
85                  throw new InvalidOperationException("В хранилище нет связей.");
86              }
87              links.Each(links.Constants.Any, links.Constants.Any, link =>
88              {
89                  firstLink = link[links.Constants.IndexPart];
90                  return links.Constants.Break;
91              });
92              if (equalityComparer.Equals(firstLink, default))
93              {
94                  throw new InvalidOperationException("В процессе поиска по хранилищу не было
                    ↪ найдено связей.");
95              }
96              return firstLink;
97          }
98
99          #region Paths
100
101         /// <remarks>
102         /// TODO: Как так? Как то что ниже может быть корректно?
103         /// Скорее всего практически не применимо
104         /// Предполагалось, что можно было конвертировать формируемый в проходе через
            ↪ SequenceWalker
105         /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
106         /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
107         /// </remarks>
108         public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ path)
109         {
110             var current = path[0];
111             //EnsureLinkExists(current, "path");
112             if (!links.Exists(current))
113             {
114                 return false;
115             }
116             var equalityComparer = EqualityComparer<TLink>.Default;
117             var constants = links.Constants;
118             for (var i = 1; i < path.Length; i++)
119             {
120                 var next = path[i];
121                 var values = links.GetLink(current);
122                 var source = values[constants.SourcePart];
123                 var target = values[constants.TargetPart];
124                 if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                    ↪ next))
125                 {
126                     //throw new InvalidOperationException(string.Format("Невозможно выбрать
                        ↪ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
127                     return false;
128                 }
129                 if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                    ↪ target))
130                 {
131                     //throw new InvalidOperationException(string.Format("Невозможно продолжить
                        ↪ путь через элемент пути {0}", next));
```

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

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

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

    #endregion

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

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

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

```
201
202         /// <summary>
203         /// Возвращает индекс конечной (Target) связи для указанной связи.
204         /// </summary>
205         /// <param name="links">Хранилище связей.</param>
206         /// <param name="link">Индекс связи.</param>
207         /// <returns>Индекс конечной связи для указанной связи.</returns>
208         [MethodImpl(MethodImplOptions.AggressiveInlining)]
209         public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ links.GetLink(link)[links.Constants.TargetPart];
210
211         /// <summary>
212         /// Возвращает индекс конечной (Target) связи для указанной связи.
213         /// </summary>
214         /// <param name="links">Хранилище связей.</param>
215         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
216         /// <returns>Индекс конечной связи для указанной связи.</returns>
217         [MethodImpl(MethodImplOptions.AggressiveInlining)]
218         public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.TargetPart];
219
220         /// <summary>
221         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
222         /// </summary>
223         /// <param name="links">Хранилище связей.</param>
224         /// <param name="handler">Обработчик каждой подходящей связи.</param>
225         /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
            ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
            ↪ Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
226         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
227         [MethodImpl(MethodImplOptions.AggressiveInlining)]
228         public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
            ↪ handler, params TLink[] restrictions)
229             => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
                ↪ links.Constants.Continue);
230
231         /// <summary>
232         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
233         /// </summary>
234         /// <param name="links">Хранилище связей.</param>
235         /// <param name="source">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
            ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
236         /// <param name="target">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
            ↪ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
237         /// <param name="handler">Обработчик каждой подходящей связи.</param>
238         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
239         [MethodImpl(MethodImplOptions.AggressiveInlining)]
240         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪ Func<TLink, bool> handler)
241         {
242             var constants = links.Constants;
243             return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
                ↪ constants.Break, constants.Any, source, target);
244         }
245
246         /// <summary>
247         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
248         /// </summary>
249         /// <param name="links">Хранилище связей.</param>
250         /// <param name="source">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
            ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
251         /// <param name="target">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
            ↪ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
252         /// <param name="handler">Обработчик каждой подходящей связи.</param>
253         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
```

```
254         [MethodImpl(MethodImplOptions.AggressiveInlining)]
255         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪  Func<IList<TLink>, TLink> handler)
256         {
257             var constants = links.Constants;
258             return links.Each(handler, constants.Any, source, target);
259         }
260
261         [MethodImpl(MethodImplOptions.AggressiveInlining)]
262         public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
            ↪  restrictions)
263         {
264             long arraySize = (Integer<TLink>)links.Count(restrictions);
265             var array = new IList<TLink>[arraySize];
266             if (arraySize > 0)
267             {
268                 var filler = new ArrayFiller<IList<TLink>, TLink>(array,
                    ↪  links.Constants.Continue);
269                 links.Each(filler.AddAndReturnConstant, restrictions);
270             }
271             return array;
272         }
273
274         [MethodImpl(MethodImplOptions.AggressiveInlining)]
275         public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
            ↪  restrictions)
276         {
277             long arraySize = (Integer<TLink>)links.Count(restrictions);
278             var array = new TLink[arraySize];
279             if (arraySize > 0)
280             {
281                 var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
282                 links.Each(filler.AddFirstAndReturnConstant, restrictions);
283             }
284             return array;
285         }
286
287         /// <summary>
288         /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
            ↪  в хранилище связей.
289         /// </summary>
290         /// <param name="links">Хранилище связей.</param>
291         /// <param name="source">Начало связи.</param>
292         /// <param name="target">Конец связи.</param>
293         /// <returns>Значение, определяющее существует ли связь.</returns>
294         [MethodImpl(MethodImplOptions.AggressiveInlining)]
295         public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
            ↪  => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
            ↪  default) > 0;
296
297         #region Ensure
298         // TODO: May be move to EnsureExtensions or make it both there and here
299
300         [MethodImpl(MethodImplOptions.AggressiveInlining)]
301         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
            ↪  reference, string argumentName)
302         {
303             if (links.Constants.IsInnerReference(reference) && !links.Exists(reference))
304             {
305                 throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
306             }
307         }
308
309         [MethodImpl(MethodImplOptions.AggressiveInlining)]
310         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
            ↪  IList<TLink> restrictions, string argumentName)
311         {
312             for (int i = 0; i < restrictions.Count; i++)
313             {
314                 links.EnsureInnerReferenceExists(restrictions[i], argumentName);
315             }
316         }
317
318         [MethodImpl(MethodImplOptions.AggressiveInlining)]
319         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
            ↪  restrictions)
320         {
321             for (int i = 0; i < restrictions.Count; i++)
```

```
            {
                links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
            }
        }

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

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

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

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

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

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

        /// <param name="links">Хранилище связей.</param>
        public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
        ↪  params TLink[] addresses)
        {
            var constants = links.Constants;

            var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
            ↪  !links.Exists(x)));
            if (nonExistentAddresses.Count > 0)
            {
                var max = nonExistentAddresses.Max();
                max = (Integer<TLink>)System.Math.Min((ulong)(Integer<TLink>)max,
                ↪  (ulong)(Integer<TLink>)constants.PossibleInnerReferencesRange.Maximum);
                var createdLinks = new List<TLink>();
                var equalityComparer = EqualityComparer<TLink>.Default;
                TLink createdLink = creator();
                while (!equalityComparer.Equals(createdLink, max))
                {
                    createdLinks.Add(createdLink);
                }
                for (var i = 0; i < createdLinks.Count; i++)
                {
                    if (!nonExistentAddresses.Contains(createdLinks[i]))
```

```
                {
                    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);
    }

```

```
462         /// <param name="links">Хранилище связей.</param>
463         [MethodImpl(MethodImplOptions.AggressiveInlining)]
464         public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target) => links.Update(links.Create(), source, target);
465
466         /// <summary>
467         /// Обновляет связь с указанными началом (Source) и концом (Target)
468         /// на связь с указанными началом (NewSource) и концом (NewTarget).
469         /// </summary>
470         /// <param name="links">Хранилище связей.</param>
471         /// <param name="link">Индекс обновляемой связи.</param>
472         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪ выполняется обновление.</param>
473         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪ выполняется обновление.</param>
474         /// <returns>Индекс обновлённой связи.</returns>
475         [MethodImpl(MethodImplOptions.AggressiveInlining)]
476         public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
            ↪ TLink newTarget) => links.Update(new LinkAddress<TLink>(link), new Link<TLink>(link,
            ↪ newSource, newTarget));
477
478         /// <summary>
479         /// Обновляет связь с указанными началом (Source) и концом (Target)
480         /// на связь с указанными началом (NewSource) и концом (NewTarget).
481         /// </summary>
482         /// <param name="links">Хранилище связей.</param>
483         /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
            ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
            ↪ Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
            ↪ связи.</param>
484         /// <returns>Индекс обновлённой связи.</returns>
485         [MethodImpl(MethodImplOptions.AggressiveInlining)]
486         public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
487         {
488             if (restrictions.Length == 2)
489             {
490                 return links.MergeAndDelete(restrictions[0], restrictions[1]);
491             }
492             if (restrictions.Length == 4)
493             {
494                 return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
                    ↪ restrictions[2], restrictions[3]);
495             }
496             else
497             {
498                 return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
499             }
500         }
501
502         [MethodImpl(MethodImplOptions.AggressiveInlining)]
503         public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
            ↪ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
504         {
505             var equalityComparer = EqualityComparer<TLink>.Default;
506             var constants = links.Constants;
507             var restrictionsIndex = restrictions[constants.IndexPart];
508             var substitutionIndex = substitution[constants.IndexPart];
509             if (equalityComparer.Equals(substitutionIndex, default))
510             {
511                 substitutionIndex = restrictionsIndex;
512             }
513             var source = substitution[constants.SourcePart];
514             var target = substitution[constants.TargetPart];
515             source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
516             target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
517             return new Link<TLink>(substitutionIndex, source, target);
518         }
519
520         /// <summary>
521         /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
            ↪ с указанными Source (началом) и Target (концом).
522         /// </summary>
523         /// <param name="links">Хранилище связей.</param>
524         /// <param name="source">Индекс связи, которая является началом на создаваемой
            ↪ связи.</param>
525         /// <param name="target">Индекс связи, которая является концом для создаваемой
            ↪ связи.</param>
```

```
526         /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
527         [MethodImpl(MethodImplOptions.AggressiveInlining)]
528         public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
529         {
530             var link = links.SearchOrDefault(source, target);
531             if (EqualityComparer<TLink>.Default.Equals(link, default))
532             {
533                 link = links.CreateAndUpdate(source, target);
534             }
535             return link;
536         }
537 
538         /// <summary>
539         /// Обновляет связь с указанными началом (Source) и концом (Target)
540         /// на связь с указанными началом (NewSource) и концом (NewTarget).
541         /// </summary>
542         /// <param name="links">Хранилище связей.</param>
543         /// <param name="source">Индекс связи, которая является началом обновляемой
            ↪ связи.</param>
544         /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
545         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪ выполняется обновление.</param>
546         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪ выполняется обновление.</param>
547         /// <returns>Индекс обновлённой связи.</returns>
548         [MethodImpl(MethodImplOptions.AggressiveInlining)]
549         public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
            ↪ TLink target, TLink newSource, TLink newTarget)
550         {
551             var equalityComparer = EqualityComparer<TLink>.Default;
552             var link = links.SearchOrDefault(source, target);
553             if (equalityComparer.Equals(link, default))
554             {
555                 return links.CreateAndUpdate(newSource, newTarget);
556             }
557             if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
                ↪ target))
558             {
559                 return link;
560             }
561             return links.Update(link, newSource, newTarget);
562         }
563 
564         /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
565         /// <param name="links">Хранилище связей.</param>
566         /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
567         /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
568         [MethodImpl(MethodImplOptions.AggressiveInlining)]
569         public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
570         {
571             var link = links.SearchOrDefault(source, target);
572             if (!EqualityComparer<TLink>.Default.Equals(link, default))
573             {
574                 links.Delete(link);
575                 return link;
576             }
577             return default;
578         }
579 
580         /// <summary>Удаляет несколько связей.</summary>
581         /// <param name="links">Хранилище связей.</param>
582         /// <param name="deletedLinks">Список адресов связей к удалению.</param>
583         [MethodImpl(MethodImplOptions.AggressiveInlining)]
584         public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
585         {
586             for (int i = 0; i < deletedLinks.Count; i++)
587             {
588                 links.Delete(deletedLinks[i]);
589             }
590         }
591 
592         /// <remarks>Before execution of this method ensure that deleted link is detached (all
            ↪ values - source and target are reset to null) or it might enter into infinite
            ↪ recursion.</remarks>
593         public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
594         {
```

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Incrementers
{
    public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
```

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

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

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

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

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

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

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

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

        private readonly TLink _unaryOne;

        public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
        ↪ _unaryOne = unaryOne;

        public TLink Increment(TLink unaryNumber)
        {
            if (_equalityComparer.Equals(unaryNumber, _unaryOne))
            {
                return Links.GetOrCreate(_unaryOne, _unaryOne);
            }
            var source = Links.GetSource(unaryNumber);
            var target = Links.GetTarget(unaryNumber);
            if (_equalityComparer.Equals(source, target))
            {
                return Links.GetOrCreate(unaryNumber, _unaryOne);
            }
            else
            {
                return Links.GetOrCreate(source, Increment(target));
            }
        }
    }
}
```

./Platform.Data.Doublets/ISynchronizedLinks.cs

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

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

./Platform.Data.Doublets/Link.cs

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

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

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

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

        private const int Length = 3;

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

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

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

        public Link(TLink index, TLink source, TLink target)
        {
            Index = index;
            Source = source;
            Target = target;
        }

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

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

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

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

        public override bool Equals(object other) => other is Link<TLink> && Equals((Link<TLink>)other);

        public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
                                              && _equalityComparer.Equals(Source, other.Source)
                                              && _equalityComparer.Equals(Target, other.Target);
```

```
70
71          public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
            ↪ {source}->{target})";
72
73          public static string ToString(TLink source, TLink target) => $"({source}->{target})";
74
75          public static implicit operator TLink[](Link<TLink> link) => link.ToArray();
76
77          public static implicit operator Link<TLink>(TLink[] linkArray) => new
            ↪ Link<TLink>(linkArray);
78
79          public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
            ↪ ToString(Source, Target) : ToString(Index, Source, Target);
80
81          #region IList
82
83          public int Count => Length;
84
85          public bool IsReadOnly => true;
86
87          public TLink this[int index]
88          {
89              get
90              {
91                  Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
                    ↪ nameof(index));
92                  if (index == _constants.IndexPart)
93                  {
94                      return Index;
95                  }
96                  if (index == _constants.SourcePart)
97                  {
98                      return Source;
99                  }
100                 if (index == _constants.TargetPart)
101                 {
102                     return Target;
103                 }
104                 throw new NotSupportedException(); // Impossible path due to
                    ↪ Ensure.ArgumentInRange
105             }
106             set => throw new NotSupportedException();
107         }
108
109         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
110
111         public IEnumerator<TLink> GetEnumerator()
112         {
113             yield return Index;
114             yield return Source;
115             yield return Target;
116         }
117
118         public void Add(TLink item) => throw new NotSupportedException();
119
120         public void Clear() => throw new NotSupportedException();
121
122         public bool Contains(TLink item) => IndexOf(item) >= 0;
123
124         public void CopyTo(TLink[] array, int arrayIndex)
125         {
126             Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
127             Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                ↪ nameof(arrayIndex));
128             if (arrayIndex + Length > array.Length)
129             {
130                 throw new InvalidOperationException();
131             }
132             array[arrayIndex++] = Index;
133             array[arrayIndex++] = Source;
134             array[arrayIndex] = Target;
135         }
136
137         public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
138
139         public int IndexOf(TLink item)
140         {
141             if (_equalityComparer.Equals(Index, item))
142             {
```

```
                return _constants.IndexPart;
            }
            if (_equalityComparer.Equals(Source, item))
            {
                return _constants.SourcePart;
            }
            if (_equalityComparer.Equals(Target, item))
            {
                return _constants.TargetPart;
            }
            return -1;
        }

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

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

        #endregion
    }
}
```

./Platform.Data.Doublets/LinkExtensions.cs

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

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

./Platform.Data.Doublets/LinksOperatorBase.cs

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

namespace Platform.Data.Doublets
{
    public abstract class LinksOperatorBase<TLink>
    {
        public ILinks<TLink> Links { get; }
        protected LinksOperatorBase(ILinks<TLink> links) => Links = links;
    }
}
```

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

```
using Platform.Interfaces;

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

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

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

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

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

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

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

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

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

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

        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;

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

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

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

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

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

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

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

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

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

```
            {
                return default;
            }
            var frequencyNumber = Links.GetSource(frequency);
            return _unaryNumberToAddressConverter.Convert(frequencyNumber);
        }
    }
}
```

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

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

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

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

        private readonly TLink[] _unaryNumberPowersOf2;

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

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

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

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

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

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

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

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

        private void InitUnaryToUInt64()
        {
            var one = Integer<TLink>.One;
            _unaryToUInt64 = new Dictionary<TLink, TLink>
            {
```

```
                { _unaryOne, one }
            };
            var unary = _unaryOne;
            var number = one;
            for (var i = 1; i < 64; i++)
            {
                unary = Links.GetOrCreate(unary, unary);
                number = Double(number);
                _unaryToUInt64.Add(unary, number);
            }
        }

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

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

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

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

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

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

        private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;

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

```

```
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
```

```
{
    public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IPropertyOperator<TLink,
    ↪ TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;

        private readonly TLink _propertyMarker;
        private readonly TLink _propertyValueMarker;

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

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

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

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

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

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

```
#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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
```

```
51          {
52              ref var firstLink = ref GetLinkReference(first);
53              ref var secondLink = ref GetLinkReference(second);
54              return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
55          }
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
59          {
60              ref var firstLink = ref GetLinkReference(first);
61              ref var secondLink = ref GetLinkReference(second);
62              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
63          }
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          protected static TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5, -5);
67
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          protected static void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
            ↪  Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
70
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          protected bool GetLeftIsChildValue(TLink value)
73          {
74              unchecked
75              {
76                  //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
77                  return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
78              }
79          }
80
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          protected static void SetLeftIsChildValue(ref TLink storedValue, bool value)
83          {
84              unchecked
85              {
86                  var previousValue = storedValue;
87                  var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
                    ↪  1);
88                  storedValue = modified;
89              }
90          }
91
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          protected bool GetRightIsChildValue(TLink value)
94          {
95              unchecked
96              {
97                  //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
98                  return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
99              }
100         }
101
102         [MethodImpl(MethodImplOptions.AggressiveInlining)]
103         protected static void SetRightIsChildValue(ref TLink storedValue, bool value)
104         {
105             unchecked
106             {
107                 var previousValue = storedValue;
108                 var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
                    ↪  1);
109                 storedValue = modified;
110             }
111         }
112
113         [MethodImpl(MethodImplOptions.AggressiveInlining)]
114         protected static sbyte GetBalanceValue(TLink storedValue)
115         {
116             unchecked
117             {
118                 var value = (int)(Integer<TLink>)Bit<TLink>.PartialRead(storedValue, 0, 3);
119                 value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
                    ↪  end of sbyte
120                 return (sbyte)value;
121             }
122         }
123
```

```
124             [MethodImpl(MethodImplOptions.AggressiveInlining)]
125             protected static void SetBalanceValue(ref TLink storedValue, sbyte value)
126             {
127                 unchecked
128                 {
129                     var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
                        ↪ 3);
130                     var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
131                     storedValue = modified;
132                 }
133             }
134 
135             public TLink this[TLink index]
136             {
137                 get
138                 {
139                     var root = GetTreeRoot();
140                     if (GreaterOrEqualThan(index, GetSize(root)))
141                     {
142                         return Zero;
143                     }
144                     while (!EqualToZero(root))
145                     {
146                         var left = GetLeftOrDefault(root);
147                         var leftSize = GetSizeOrZero(left);
148                         if (LessThan(index, leftSize))
149                         {
150                             root = left;
151                             continue;
152                         }
153                         if (IsEquals(index, leftSize))
154                         {
155                             return root;
156                         }
157                         root = GetRightOrDefault(root);
158                         index = Subtract(index, Increment(leftSize));
159                     }
160                     return Zero; // TODO: Impossible situation exception (only if tree structure
                        ↪ broken)
161                 }
162             }
163 
164             /// <summary>
165             /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪ (концом).
166             /// </summary>
167             /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
168             /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
169             /// <returns>Индекс искомой связи.</returns>
170             public TLink Search(TLink source, TLink target)
171             {
172                 var root = GetTreeRoot();
173                 while (!EqualToZero(root))
174                 {
175                     ref var rootLink = ref GetLinkReference(root);
176                     var rootSource = rootLink.Source;
177                     var rootTarget = rootLink.Target;
178                     if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                        ↪ node.Key < root.Key
179                     {
180                         root = GetLeftOrDefault(root);
181                     }
182                     else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                        ↪ node.Key > root.Key
183                     {
184                         root = GetRightOrDefault(root);
185                     }
186                     else // node.Key == root.Key
187                     {
188                         return root;
189                     }
190                 }
191                 return Zero;
192             }
193 
194             // TODO: Return indices range instead of references count
195             public TLink CountUsages(TLink link)
196             {
```

```
            var root = GetTreeRoot();
            var total = GetSize(root);
            var totalRightIgnore = Zero;
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (LessOrEqualThan(@base, link))
                {
                    root = GetRightOrDefault(root);
                }
                else
                {
                    totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
                    root = GetLeftOrDefault(root);
                }
            }
            root = GetTreeRoot();
            var totalLeftIgnore = Zero;
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (GreaterOrEqualThan(@base, link))
                {
                    root = GetLeftOrDefault(root);
                }
                else
                {
                    totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));

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

        public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
        {
            var root = GetTreeRoot();
            if (EqualToZero(root))
            {
                return _constants.Continue;
            }
            TLink first = Zero, current = root;
            while (!EqualToZero(current))
            {
                var @base = GetBasePartValue(current);
                if (GreaterOrEqualThan(@base, link))
                {
                    if (IsEquals(@base, link))
                    {
                        first = current;
                    }
                    current = GetLeftOrDefault(current);
                }
                else
                {
                    current = GetRightOrDefault(current);
                }
            }
            if (!EqualToZero(first))
            {
                current = first;
                while (true)
                {
                    if (IsEquals(handler(GetLinkValues(current)), _constants.Break))
                    {
                        return _constants.Break;
                    }
                    current = GetNext(current);
                    if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
                    {
                        break;
                    }
                }
            }
            return _constants.Continue;
        }

        protected override void PrintNodeValue(TLink node, StringBuilder sb)
```

```
        {
            ref var link = ref GetLinkReference(node);
            sb.Append(' ');
            sb.Append(link.Source);
            sb.Append('-');
            sb.Append('>');
            sb.Append(link.Target);
        }
    }
}
```

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

```
using Platform.Unsafe;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override void SetLeft(TLink node, TLink left) =>
        ↪  GetLinkReference(node).LeftAsTarget = left;

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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/RawLink.cs

```
using Platform.Unsafe;

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

        public TLink Source;
        public TLink Target;
        public TLink LeftAsSource;
        public TLink RightAsSource;
```

```
        public TLink SizeAsSource;
        public TLink LeftAsTarget;
        public TLink RightAsTarget;
        public TLink SizeAsTarget;
    }
}
```

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

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

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

// ReSharper disable StaticMemberInGenericType
// ReSharper disable BuiltInTypeReferenceStyle
// ReSharper disable MemberCanBePrivate.Local
// ReSharper disable UnusedMember.Local

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public unsafe partial class ResizableDirectMemoryLinks<TLink> : DisposableBase, ILinks<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        /// <summary>Возвращает размер одной связи в байтах.</summary>
        public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;

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

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

        private readonly long _memoryReservationStep;

        private readonly IResizableDirectMemory _memory;
        private byte* _header;
        private byte* _links;

        private ILinksTreeMethods<TLink> _targetsTreeMethods;
        private ILinksTreeMethods<TLink> _sourcesTreeMethods;

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

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

        public LinksConstants<TLink> Constants { get; }

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

        /// <summary>
        /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным минимальным шагом расширения базы данных.
        /// </summary>
        /// <param name="address">Полный пусть к файлу базы данных.</param>
```

```
69          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪  байтах.</param>
70          public ResizableDirectMemoryLinks(string address, long memoryReservationStep) : this(new
            ↪  FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪  memoryReservationStep) { }
71
72          public ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪  DefaultLinksSizeStep) { }
73
74          public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep)
75          {
76              Constants = Default<LinksConstants<TLink>>.Instance;
77              _memory = memory;
78              _memoryReservationStep = memoryReservationStep;
79              if (memory.ReservedCapacity < memoryReservationStep)
80              {
81                  memory.ReservedCapacity = memoryReservationStep;
82              }
83              SetPointers(_memory);
84              ref var header = ref AsRef<LinksHeader<TLink>>(_header);
85              // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
86              _memory.UsedCapacity = ((Integer<TLink>)header.AllocatedLinks * LinkSizeInBytes) +
                ↪  LinkHeaderSizeInBytes;
87              // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
88              header.ReservedLinks = (Integer<TLink>)((_memory.ReservedCapacity -
                ↪  LinkHeaderSizeInBytes) / LinkSizeInBytes);
89          }
90
91          [MethodImpl(MethodImplOptions.AggressiveInlining)]
92          public TLink Count(IList<TLink> restrictions)
93          {
94              // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
95              if (restrictions.Count == 0)
96              {
97                  return Total;
98              }
99              if (restrictions.Count == 1)
100             {
101                 var index = restrictions[Constants.IndexPart];
102                 if (_equalityComparer.Equals(index, Constants.Any))
103                 {
104                     return Total;
105                 }
106                 return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
107             }
108             if (restrictions.Count == 2)
109             {
110                 var index = restrictions[Constants.IndexPart];
111                 var value = restrictions[1];
112                 if (_equalityComparer.Equals(index, Constants.Any))
113                 {
114                     if (_equalityComparer.Equals(value, Constants.Any))
115                     {
116                         return Total; // Any - как отсутствие ограничения
117                     }
118                     return Add(_sourcesTreeMethods.CountUsages(value),
                        ↪  _targetsTreeMethods.CountUsages(value));
119                 }
120                 else
121                 {
122                     if (!Exists(index))
123                     {
124                         return Integer<TLink>.Zero;
125                     }
126                     if (_equalityComparer.Equals(value, Constants.Any))
127                     {
128                         return Integer<TLink>.One;
129                     }
130                     ref var storedLinkValue = ref GetLinkUnsafe(index);
131                     if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
132                         _equalityComparer.Equals(storedLinkValue.Target, value))
133                     {
134                         return Integer<TLink>.One;
135                     }
136                     return Integer<TLink>.Zero;
137                 }
138             }
```

```
139             if (restrictions.Count == 3)
140             {
141                 var index = restrictions[Constants.IndexPart];
142                 var source = restrictions[Constants.SourcePart];
143                 var target = restrictions[Constants.TargetPart];
144 
145                 if (_equalityComparer.Equals(index, Constants.Any))
146                 {
147                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ _equalityComparer.Equals(target, Constants.Any))
148                     {
149                         return Total;
150                     }
151                     else if (_equalityComparer.Equals(source, Constants.Any))
152                     {
153                         return _targetsTreeMethods.CountUsages(target);
154                     }
155                     else if (_equalityComparer.Equals(target, Constants.Any))
156                     {
157                         return _sourcesTreeMethods.CountUsages(source);
158                     }
159                     else //if(source != Any && target != Any)
160                     {
161                         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
162                         var link = _sourcesTreeMethods.Search(source, target);
163                         return _equalityComparer.Equals(link, Constants.Null) ?
                            ↪ Integer<TLink>.Zero : Integer<TLink>.One;
164                     }
165                 }
166                 else
167                 {
168                     if (!Exists(index))
169                     {
170                         return Integer<TLink>.Zero;
171                     }
172                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ _equalityComparer.Equals(target, Constants.Any))
173                     {
174                         return Integer<TLink>.One;
175                     }
176                     ref var storedLinkValue = ref GetLinkUnsafe(index);
177                     if (!_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ !_equalityComparer.Equals(target, Constants.Any))
178                     {
179                         if (_equalityComparer.Equals(storedLinkValue.Source, source) &&
180                             _equalityComparer.Equals(storedLinkValue.Target, target))
181                         {
182                             return Integer<TLink>.One;
183                         }
184                         return Integer<TLink>.Zero;
185                     }
186                     var value = default(TLink);
187                     if (_equalityComparer.Equals(source, Constants.Any))
188                     {
189                         value = target;
190                     }
191                     if (_equalityComparer.Equals(target, Constants.Any))
192                     {
193                         value = source;
194                     }
195                     if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
196                         _equalityComparer.Equals(storedLinkValue.Target, value))
197                     {
198                         return Integer<TLink>.One;
199                     }
200                     return Integer<TLink>.Zero;
201                 }
202             }
203             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪ поддерживаются.");
204         }
205 
206         [MethodImpl(MethodImplOptions.AggressiveInlining)]
207         public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
208         {
209             if (restrictions.Count == 0)
210             {
```

```
211                 for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
                    ↪  (Integer<TLink>)AsRef<LinksHeader<TLink>>(_header).AllocatedLinks) <= 0;
                    ↪  link = Increment(link))
212                 {
213                     if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
                        ↪  Constants.Break))
214                     {
215                         return Constants.Break;
216                     }
217                 }
218                 return Constants.Continue;
219             }
220             if (restrictions.Count == 1)
221             {
222                 var index = restrictions[Constants.IndexPart];
223                 if (_equalityComparer.Equals(index, Constants.Any))
224                 {
225                     return Each(handler, ArrayPool<TLink>.Empty);
226                 }
227                 if (!Exists(index))
228                 {
229                     return Constants.Continue;
230                 }
231                 return handler(GetLinkStruct(index));
232             }
233             if (restrictions.Count == 2)
234             {
235                 var index = restrictions[Constants.IndexPart];
236                 var value = restrictions[1];
237                 if (_equalityComparer.Equals(index, Constants.Any))
238                 {
239                     if (_equalityComparer.Equals(value, Constants.Any))
240                     {
241                         return Each(handler, ArrayPool<TLink>.Empty);
242                     }
243                     if (_equalityComparer.Equals(Each(handler, new[] { index, value,
                        ↪  Constants.Any }), Constants.Break))
244                     {
245                         return Constants.Break;
246                     }
247                     return Each(handler, new[] { index, Constants.Any, value });
248                 }
249                 else
250                 {
251                     if (!Exists(index))
252                     {
253                         return Constants.Continue;
254                     }
255                     if (_equalityComparer.Equals(value, Constants.Any))
256                     {
257                         return handler(GetLinkStruct(index));
258                     }
259                     ref var storedLinkValue = ref GetLinkUnsafe(index);
260                     if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
261                         _equalityComparer.Equals(storedLinkValue.Target, value))
262                     {
263                         return handler(GetLinkStruct(index));
264                     }
265                     return Constants.Continue;
266                 }
267             }
268             if (restrictions.Count == 3)
269             {
270                 var index = restrictions[Constants.IndexPart];
271                 var source = restrictions[Constants.SourcePart];
272                 var target = restrictions[Constants.TargetPart];
273                 if (_equalityComparer.Equals(index, Constants.Any))
274                 {
275                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪  _equalityComparer.Equals(target, Constants.Any))
276                     {
277                         return Each(handler, ArrayPool<TLink>.Empty);
278                     }
279                     else if (_equalityComparer.Equals(source, Constants.Any))
280                     {
281                         return _targetsTreeMethods.EachUsage(target, handler);
282                     }
283                     else if (_equalityComparer.Equals(target, Constants.Any))
```

```
284                     {
285                         return _sourcesTreeMethods.EachUsage(source, handler);
286                     }
287                     else //if(source != Any && target != Any)
288                     {
289                         var link = _sourcesTreeMethods.Search(source, target);
290                         return _equalityComparer.Equals(link, Constants.Null) ?
                            ↪  Constants.Continue : handler(GetLinkStruct(link));
291                     }
292                 }
293                 else
294                 {
295                     if (!Exists(index))
296                     {
297                         return Constants.Continue;
298                     }
299                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪  _equalityComparer.Equals(target, Constants.Any))
300                     {
301                         return handler(GetLinkStruct(index));
302                     }
303                     ref var storedLinkValue = ref GetLinkUnsafe(index);
304                     if (!_equalityComparer.Equals(source, Constants.Any) &&
                        ↪  !_equalityComparer.Equals(target, Constants.Any))
305                     {
306                         if (_equalityComparer.Equals(storedLinkValue.Source, source) &&
307                             _equalityComparer.Equals(storedLinkValue.Target, target))
308                         {
309                             return handler(GetLinkStruct(index));
310                         }
311                         return Constants.Continue;
312                     }
313                     var value = default(TLink);
314                     if (_equalityComparer.Equals(source, Constants.Any))
315                     {
316                         value = target;
317                     }
318                     if (_equalityComparer.Equals(target, Constants.Any))
319                     {
320                         value = source;
321                     }
322                     if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
323                         _equalityComparer.Equals(storedLinkValue.Target, value))
324                     {
325                         return handler(GetLinkStruct(index));
326                     }
327                     return Constants.Continue;
328                 }
329             }
330             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪  поддерживаются.");
331         }
332 
333         /// <remarks>
334         /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
                ↪  в другом месте (но не в менеджере памяти, а в логике Links)
335         /// </remarks>
336         [MethodImpl(MethodImplOptions.AggressiveInlining)]
337         public TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
338         {
339             var linkIndex = restrictions[Constants.IndexPart];
340             ref var link = ref GetLinkUnsafe(linkIndex);
341             ref var firstAsSource = ref AsRef<LinksHeader<TLink>>(_header).FirstAsSource;
342             ref var firstAsTarget = ref AsRef<LinksHeader<TLink>>(_header).FirstAsTarget;
343             // Будет корректно работать только в том случае, если пространство выделенной связи
                ↪  предварительно заполнено нулями
344             if (!_equalityComparer.Equals(link.Source, Constants.Null))
345             {
346                 _sourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
347             }
348             if (!_equalityComparer.Equals(link.Target, Constants.Null))
349             {
350                 _targetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
351             }
352             link.Source = substitution[Constants.SourcePart];
353             link.Target = substitution[Constants.TargetPart];
354             if (!_equalityComparer.Equals(link.Source, Constants.Null))
355             {
```

```
356                 _sourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
357             }
358             if (!_equalityComparer.Equals(link.Target, Constants.Null))
359             {
360                 _targetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
361             }
362             return linkIndex;
363         }
364 
365         [MethodImpl(MethodImplOptions.AggressiveInlining)]
366         public Link<TLink> GetLinkStruct(TLink linkIndex)
367         {
368             ref var link = ref GetLinkUnsafe(linkIndex);
369             return new Link<TLink>(linkIndex, link.Source, link.Target);
370         }
371 
372         [MethodImpl(MethodImplOptions.AggressiveInlining)]
373         internal ref RawLink<TLink> GetLinkUnsafe(TLink linkIndex) => ref
             ↪  AsRef<RawLink<TLink>>(_links + LinkSizeInBytes * (Integer<TLink>)linkIndex);
374 
375         /// <remarks>
376         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
             ↪  пространство
377         /// </remarks>
378         public TLink Create(IList<TLink> restrictions)
379         {
380             ref var header = ref AsRef<LinksHeader<TLink>>(_header);
381             var freeLink = header.FirstFreeLink;
382             if (!_equalityComparer.Equals(freeLink, Constants.Null))
383             {
384                 _unusedLinksListMethods.Detach(freeLink);
385             }
386             else
387             {
388                 var maximumPossibleInnerReference =
                     ↪  Constants.PossibleInnerReferencesRange.Maximum;
389                 if (_comparer.Compare(header.AllocatedLinks, maximumPossibleInnerReference) > 0)
390                 {
391                     throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
392                 }
393                 if (_comparer.Compare(header.AllocatedLinks, Decrement(header.ReservedLinks)) >=
                     ↪  0)
394                 {
395                     _memory.ReservedCapacity += _memoryReservationStep;
396                     SetPointers(_memory);
397                     header.ReservedLinks = (Integer<TLink>)(_memory.ReservedCapacity /
                         ↪  LinkSizeInBytes);
398                 }
399                 header.AllocatedLinks = Increment(header.AllocatedLinks);
400                 _memory.UsedCapacity += LinkSizeInBytes;
401                 freeLink = header.AllocatedLinks;
402             }
403             return freeLink;
404         }
405 
406         public void Delete(IList<TLink> restrictions)
407         {
408             ref var header = ref AsRef<LinksHeader<TLink>>(_header);
409             var link = restrictions[Constants.IndexPart];
410             if (_comparer.Compare(link, header.AllocatedLinks) < 0)
411             {
412                 _unusedLinksListMethods.AttachAsFirst(link);
413             }
414             else if (_equalityComparer.Equals(link, header.AllocatedLinks))
415             {
416                 header.AllocatedLinks = Decrement(header.AllocatedLinks);
417                 _memory.UsedCapacity -= LinkSizeInBytes;
418                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                     ↪  пока не дойдём до первой существующей связи
419                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
420                 while ((_comparer.Compare(header.AllocatedLinks, Integer<TLink>.Zero) > 0) &&
                     ↪  IsUnusedLink(header.AllocatedLinks))
421                 {
422                     _unusedLinksListMethods.Detach(header.AllocatedLinks);
423                     header.AllocatedLinks = Decrement(header.AllocatedLinks);
424                     _memory.UsedCapacity -= LinkSizeInBytes;
425                 }
426             }
```

```
427         }
428
429         /// <remarks>
430         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
            ↪  адрес реально поменялся
431         ///
432         /// Указатель this.links может быть в том же месте,
433         /// так как 0-я связь не используется и имеет такой же размер как Header,
434         /// поэтому header размещается в том же месте, что и 0-я связь
435         /// </remarks>
436         private void SetPointers(IDirectMemory memory)
437         {
438             if (memory == null)
439             {
440                 _links = null;
441                 _header = _links;
442                 _unusedLinksListMethods = null;
443                 _targetsTreeMethods = null;
444                 _unusedLinksListMethods = null;
445             }
446             else
447             {
448                 _links = (byte*)(void*)memory.Pointer;
449                 _header = _links;
450                 _sourcesTreeMethods = new LinksSourcesAVLBalancedTreeMethods<TLink>(this,
                    ↪  _links, _header);
451                 _targetsTreeMethods = new LinksTargetsAVLBalancedTreeMethods<TLink>(this,
                    ↪  _links, _header);
452                 _unusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
453             }
454         }
455
456         [MethodImpl(MethodImplOptions.AggressiveInlining)]
457         private bool Exists(TLink link)
458             => (_comparer.Compare(link, Constants.PossibleInnerReferencesRange.Minimum) >= 0)
459             && (_comparer.Compare(link, AsRef<LinksHeader<TLink>>(_header).AllocatedLinks) <= 0)
460             && !IsUnusedLink(link);
461
462         [MethodImpl(MethodImplOptions.AggressiveInlining)]
463         private bool IsUnusedLink(TLink link)
464             => _equalityComparer.Equals(AsRef<LinksHeader<TLink>>(_header).FirstFreeLink, link)
465             || (_equalityComparer.Equals(GetLinkUnsafe(link).SizeAsSource, Constants.Null)
466             && !_equalityComparer.Equals(GetLinkUnsafe(link).Source, Constants.Null));
467
468         #region DisposableBase
469
470         protected override bool AllowMultipleDisposeCalls => true;
471
472         protected override void Dispose(bool manual, bool wasDisposed)
473         {
474             if (!wasDisposed)
475             {
476                 SetPointers(null);
477                 _memory.DisposeIfPossible();
478             }
479         }
480
481         #endregion
482     }
483 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using System.Text;
5  using Platform.Collections.Methods.Trees;
6  using static System.Runtime.CompilerServices.Unsafe;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory
11 {
12     public unsafe abstract class UInt64LinksAVLBalancedTreeMethodsBase :
       ↪  SizedAndThreadedAVLBalancedTreeMethods<ulong>
13     {
14         private readonly UInt64ResizableDirectMemoryLinks _memory;
15         private readonly LinksConstants<ulong> _constants;
16         internal readonly UInt64RawLink* _links;
17         internal readonly UInt64LinksHeader* _header;
```

```
18
19          internal UInt64LinksAVLBalancedTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory,
            ↪  UInt64RawLink* links, UInt64LinksHeader* header)
20          {
21              _links = links;
22              _header = header;
23              _memory = memory;
24              _constants = memory.Constants;
25          }
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override ulong GetZero() => 0UL;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override bool EqualToZero(ulong value) => value == 0UL;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override bool IsEquals(ulong first, ulong second) => first == second;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override bool GreaterThanZero(ulong value) => value > 0UL;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override bool GreaterThan(ulong first, ulong second) => first > second;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
            ↪  always true for ulong
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
            ↪  always >= 0 for ulong
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
            ↪  for ulong
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          protected override bool LessThan(ulong first, ulong second) => first < second;
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected override ulong Increment(ulong value) => ++value;
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected override ulong Decrement(ulong value) => --value;
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected override ulong Add(ulong first, ulong second) => first + second;
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override ulong Subtract(ulong first, ulong second) => first - second;
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected abstract ulong GetTreeRoot();
74
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
76          protected abstract ulong GetBasePartValue(ulong link);
77
78          [MethodImpl(MethodImplOptions.AggressiveInlining)]
79          protected abstract bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget);
80
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          protected abstract bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget);
83
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
86          {
87              ref var firstLink = ref _links[first];
88              ref var secondLink = ref _links[second];
89              return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
```

```
90          }
91
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
94          {
95              ref var firstLink = ref _links[first];
96              ref var secondLink = ref _links[second];
97              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
98          }
99
100         [MethodImpl(MethodImplOptions.AggressiveInlining)]
101         protected static ulong GetSizeValue(ulong value) => unchecked((value & 4294967264UL) >>
            ↪  5);
102
103         [MethodImpl(MethodImplOptions.AggressiveInlining)]
104         protected static void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
            ↪  unchecked((storedValue & 31UL) | ((size & 134217727UL) << 5));
105
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected static bool GetLeftIsChildValue(ulong value) => unchecked((value & 16UL) >> 4
            ↪  == 1UL);
108
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         protected static void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
            ↪  storedValue = unchecked((storedValue & 4294967279UL) | ((As<bool, byte>(ref value) &
            ↪  1UL) << 4));
111
112         [MethodImpl(MethodImplOptions.AggressiveInlining)]
113         protected static bool GetRightIsChildValue(ulong value) => unchecked((value & 8UL) >> 3
            ↪  == 1UL);
114
115         [MethodImpl(MethodImplOptions.AggressiveInlining)]
116         protected static void SetRightIsChildValue(ref ulong storedValue, bool value) =>
            ↪  storedValue = unchecked((storedValue & 4294967287UL) | ((As<bool, byte>(ref value) &
            ↪  1UL) << 3));
117
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
119         protected static sbyte GetBalanceValue(ulong value) => unchecked((sbyte)((value & 7UL) |
            ↪  0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
            ↪  sbyte
120
121         [MethodImpl(MethodImplOptions.AggressiveInlining)]
122         protected static void SetBalanceValue(ref ulong storedValue, sbyte value) => storedValue
            ↪  = unchecked((storedValue & 4294967288UL) | ((ulong)((((byte)value >> 5) & 4) | value
            ↪  & 3) & 7UL));
123
124         public ulong this[ulong index]
125         {
126             get
127             {
128                 var root = GetTreeRoot();
129                 if (index >= GetSize(root))
130                 {
131                     return 0UL;
132                 }
133                 while (root != 0UL)
134                 {
135                     var left = GetLeftOrDefault(root);
136                     var leftSize = GetSizeOrZero(left);
137                     if (index < leftSize)
138                     {
139                         root = left;
140                         continue;
141                     }
142                     if (index == leftSize)
143                     {
144                         return root;
145                     }
146                     root = GetRightOrDefault(root);
147                     index -= leftSize + 1UL;
148                 }
149                 return 0UL; // TODO: Impossible situation exception (only if tree structure
                    ↪  broken)
150             }
151         }
152
153         /// <summary>
```

```
154         /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
            ↪ (концом).
155         /// </summary>
156         /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
157         /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
158         /// <returns>Индекс искомой связи.</returns>
159         public ulong Search(ulong source, ulong target)
160         {
161             var root = GetTreeRoot();
162             while (root != 0UL)
163             {
164                 var rootLink = _links[root];
165                 var rootSource = rootLink.Source;
166                 var rootTarget = rootLink.Target;
167                 if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                    ↪ node.Key < root.Key
168                 {
169                     root = GetLeftOrDefault(root);
170                 }
171                 else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                    ↪ node.Key > root.Key
172                 {
173                     root = GetRightOrDefault(root);
174                 }
175                 else // node.Key == root.Key
176                 {
177                     return root;
178                 }
179             }
180             return 0UL;
181         }
182 
183         // TODO: Return indices range instead of references count
184         public ulong CountUsages(ulong link)
185         {
186             var root = GetTreeRoot();
187             var total = GetSize(root);
188             var totalRightIgnore = 0UL;
189             while (root != 0UL)
190             {
191                 var @base = GetBasePartValue(root);
192                 if (@base <= link)
193                 {
194                     root = GetRightOrDefault(root);
195                 }
196                 else
197                 {
198                     totalRightIgnore += GetRightSize(root) + 1UL;
199                     root = GetLeftOrDefault(root);
200                 }
201             }
202             root = GetTreeRoot();
203             var totalLeftIgnore = 0UL;
204             while (root != 0UL)
205             {
206                 var @base = GetBasePartValue(root);
207                 if (@base >= link)
208                 {
209                     root = GetLeftOrDefault(root);
210                 }
211                 else
212                 {
213                     totalLeftIgnore += GetLeftSize(root) + 1UL;
214                     root = GetRightOrDefault(root);
215                 }
216             }
217             return total - totalRightIgnore - totalLeftIgnore;
218         }
219 
220         public ulong EachUsage(ulong link, Func<IList<ulong>, ulong> handler)
221         {
222             var root = GetTreeRoot();
223             if (root == 0UL)
224             {
225                 return _constants.Continue;
226             }
227             ulong first = 0UL, current = root;
228             while (current != 0UL)
229             {
```

```
                var @base = GetBasePartValue(current);
                if (@base >= link)
                {
                    if (@base == link)
                    {
                        first = current;
                    }
                    current = GetLeftOrDefault(current);
                }
                else
                {
                    current = GetRightOrDefault(current);
                }
            }
            if (first != 0UL)
            {
                current = first;
                while (true)
                {
                    if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
                    {
                        return _constants.Break;
                    }
                    current = GetNext(current);
                    if (current == 0UL || GetBasePartValue(current) != link)
                    {
                        break;
                    }
                }
            }
            return _constants.Continue;
        }

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

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

```
namespace Platform.Data.Doublets.ResizableDirectMemory
{
    internal struct UInt64LinksHeader
    {
        public ulong AllocatedLinks;
        public ulong ReservedLinks;
        public ulong FreeLinks;
        public ulong FirstFreeLink;
        public ulong FirstAsSource;
        public ulong FirstAsTarget;
        public ulong LastFreeLink;
        public ulong Reserved8;
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

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

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

```

```
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override ref ulong GetRightReference(ulong node) => ref
            ↪  _links[node].RightAsSource;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override ulong GetLeft(ulong node) => _links[node].LeftAsSource;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override ulong GetRight(ulong node) => _links[node].RightAsSource;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override void SetLeft(ulong node, ulong left) => _links[node].LeftAsSource =
            ↪  left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(ulong node, ulong right) => _links[node].RightAsSource
            ↪  = right;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override ulong GetSize(ulong node) => GetSizeValue(_links[node].SizeAsSource);
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
            ↪  _links[node].SizeAsSource, size);
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override bool GetLeftIsChild(ulong node) =>
            ↪  GetLeftIsChildValue(_links[node].SizeAsSource);
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          protected override void SetLeftIsChild(ulong node, bool value) =>
            ↪  SetLeftIsChildValue(ref _links[node].SizeAsSource, value);
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override bool GetRightIsChild(ulong node) =>
            ↪  GetRightIsChildValue(_links[node].SizeAsSource);
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          protected override void SetRightIsChild(ulong node, bool value) =>
            ↪  SetRightIsChildValue(ref _links[node].SizeAsSource, value);
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          protected override sbyte GetBalance(ulong node) =>
            ↪  GetBalanceValue(_links[node].SizeAsSource);
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
            ↪  _links[node].SizeAsSource, value);
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override ulong GetTreeRoot() => _header->FirstAsSource;
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override ulong GetBasePartValue(ulong link) => _links[link].Source;
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget)
61              => firstSource < secondSource || (firstSource == secondSource && firstTarget <
                ↪  secondTarget);
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget)
65              => firstSource > secondSource || (firstSource == secondSource && firstTarget >
                ↪  secondTarget);
66
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          protected override void ClearNode(ulong node)
69          {
70              ref UInt64RawLink link = ref _links[node];
71              link.LeftAsSource = OUL;
72              link.RightAsSource = OUL;
73              link.SizeAsSource = OUL;
74          }
75      }
76  }
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public unsafe class UInt64LinksTargetsAVLBalancedTreeMethods :
    ↪ UInt64LinksAVLBalancedTreeMethodsBase, ILinksTreeMethods<ulong>
    {
        internal UInt64LinksTargetsAVLBalancedTreeMethods(UInt64ResizableDirectMemoryLinks
        ↪ memory, UInt64RawLink* links, UInt64LinksHeader* header) : base(memory, links,
        ↪ header) { }

        //protected override IntPtr GetLeft(ulong node) => new IntPtr(&Links[node].LeftAsTarget);

        //protected override IntPtr GetRight(ulong node) => new
        ↪ IntPtr(&Links[node].RightAsTarget);

        //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;

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

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

        //protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
        ↪ size;

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Disposables;
using Platform.Collections.Arrays;
using Platform.Singletons;
using Platform.Memory;
using Platform.Data.Exceptions;

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

// ReSharper disable BuiltInTypeReferenceStyle

//#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    using id = UInt64;

    public unsafe class UInt64ResizableDirectMemoryLinks : DisposableBase, ILinks<id>
    {
        /// <summary>Возвращает размер одной связи в байтах.</summary>
        /// <remarks>
        /// Используется только во вне класса, не рекомедуется использовать внутри.
        /// Так как во вне не обязательно будет доступен unsafe C#.
        /// </remarks>
        public static readonly int LinkSizeInBytes = sizeof(UInt64RawLink);

```

```
31          public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
32
33          private readonly long _memoryReservationStep;
34
35          private readonly IResizableDirectMemory _memory;
36          private UInt64LinksHeader* _header;
37          private UInt64RawLink* _links;
38
39          private ILinksTreeMethods<id> _targetsTreeMethods;
40          private ILinksTreeMethods<id> _sourcesTreeMethods;
41
42          // 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
43          private ILinksListMethods<id> _unusedLinksListMethods;
44
45          /// <summary>
46          /// Возвращает общее число связей находящихся в хранилище.
47          /// </summary>
48          private id Total => _header->AllocatedLinks - _header->FreeLinks;
49
50          // TODO: Дать возможность переопределять в конструкторе
51          public LinksConstants<id> Constants { get; }
52
53          public UInt64ResizableDirectMemoryLinks(string address) : this(address,
            ↪  DefaultLinksSizeStep) { }
54
55          /// <summary>
56          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪  минимальным шагом расширения базы данных.
57          /// </summary>
58          /// <param name="address">Полный пусть к файлу базы данных.</param>
59          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪  байтах.</param>
60          public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
            ↪  this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪  memoryReservationStep) { }
61
62          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪  DefaultLinksSizeStep) { }
63
64          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep)
65          {
66              Constants = Default<LinksConstants<id>>.Instance;
67              _memory = memory;
68              _memoryReservationStep = memoryReservationStep;
69              if (memory.ReservedCapacity < memoryReservationStep)
70              {
71                  memory.ReservedCapacity = memoryReservationStep;
72              }
73              SetPointers(_memory);
74              // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
75              _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(UInt64RawLink)) +
                ↪  sizeof(UInt64LinksHeader);
76              // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
77              _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(UInt64LinksHeader))
                ↪  / sizeof(UInt64RawLink));
78          }
79
80          [MethodImpl(MethodImplOptions.AggressiveInlining)]
81          public id Count(IList<id> restrictions)
82          {
83              // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
84              if (restrictions.Count == 0)
85              {
86                  return Total;
87              }
88              if (restrictions.Count == 1)
89              {
90                  var index = restrictions[Constants.IndexPart];
91                  if (index == Constants.Any)
92                  {
93                      return Total;
94                  }
95                  return Exists(index) ? 1UL : 0UL;
96              }
97              if (restrictions.Count == 2)
98              {
```

```
99                  var index = restrictions[Constants.IndexPart];
100                 var value = restrictions[1];
101                 if (index == Constants.Any)
102                 {
103                     if (value == Constants.Any)
104                     {
105                         return Total; // Any - как отсутствие ограничения
106                     }
107                     return _sourcesTreeMethods.CountUsages(value)
108                          + _targetsTreeMethods.CountUsages(value);
109                 }
110                 else
111                 {
112                     if (!Exists(index))
113                     {
114                         return 0;
115                     }
116                     if (value == Constants.Any)
117                     {
118                         return 1;
119                     }
120                     var storedLinkValue = GetLinkUnsafe(index);
121                     if (storedLinkValue->Source == value ||
122                         storedLinkValue->Target == value)
123                     {
124                         return 1;
125                     }
126                     return 0;
127                 }
128             }
129             if (restrictions.Count == 3)
130             {
131                 var index = restrictions[Constants.IndexPart];
132                 var source = restrictions[Constants.SourcePart];
133                 var target = restrictions[Constants.TargetPart];
134                 if (index == Constants.Any)
135                 {
136                     if (source == Constants.Any && target == Constants.Any)
137                     {
138                         return Total;
139                     }
140                     else if (source == Constants.Any)
141                     {
142                         return _targetsTreeMethods.CountUsages(target);
143                     }
144                     else if (target == Constants.Any)
145                     {
146                         return _sourcesTreeMethods.CountUsages(source);
147                     }
148                     else //if(source != Any && target != Any)
149                     {
150                         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
151                         var link = _sourcesTreeMethods.Search(source, target);
152                         return link == Constants.Null ? 0UL : 1UL;
153                     }
154                 }
155                 else
156                 {
157                     if (!Exists(index))
158                     {
159                         return 0;
160                     }
161                     if (source == Constants.Any && target == Constants.Any)
162                     {
163                         return 1;
164                     }
165                     var storedLinkValue = GetLinkUnsafe(index);
166                     if (source != Constants.Any && target != Constants.Any)
167                     {
168                         if (storedLinkValue->Source == source &&
169                             storedLinkValue->Target == target)
170                         {
171                             return 1;
172                         }
173                         return 0;
174                     }
175                     var value = default(id);
176                     if (source == Constants.Any)
177                     {
```

```
                        value = target;
                    }
                    if (target == Constants.Any)
                    {
                        value = source;
                    }
                    if (storedLinkValue->Source == value ||
                        storedLinkValue->Target == value)
                    {
                        return 1;
                    }
                    return 0;
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
             ↪  поддерживаются.");
        }

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

```
255                         return handler(GetLinkStruct(index));
256                     }
257                     return Constants.Continue;
258                 }
259             }
260             if (restrictions.Count == 3)
261             {
262                 var index = restrictions[Constants.IndexPart];
263                 var source = restrictions[Constants.SourcePart];
264                 var target = restrictions[Constants.TargetPart];
265                 if (index == Constants.Any)
266                 {
267                     if (source == Constants.Any && target == Constants.Any)
268                     {
269                         return Each(handler, ArrayPool<ulong>.Empty);
270                     }
271                     else if (source == Constants.Any)
272                     {
273                         return _targetsTreeMethods.EachUsage(target, handler);
274                     }
275                     else if (target == Constants.Any)
276                     {
277                         return _sourcesTreeMethods.EachUsage(source, handler);
278                     }
279                     else //if(source != Any && target != Any)
280                     {
281                         var link = _sourcesTreeMethods.Search(source, target);
282                         return link == Constants.Null ? Constants.Continue :
                            ↪ handler(GetLinkStruct(link));
283                     }
284                 }
285                 else
286                 {
287                     if (!Exists(index))
288                     {
289                         return Constants.Continue;
290                     }
291                     if (source == Constants.Any && target == Constants.Any)
292                     {
293                         return handler(GetLinkStruct(index));
294                     }
295                     var storedLinkValue = GetLinkUnsafe(index);
296                     if (source != Constants.Any && target != Constants.Any)
297                     {
298                         if (storedLinkValue->Source == source &&
299                             storedLinkValue->Target == target)
300                         {
301                             return handler(GetLinkStruct(index));
302                         }
303                         return Constants.Continue;
304                     }
305                     var value = default(id);
306                     if (source == Constants.Any)
307                     {
308                         value = target;
309                     }
310                     if (target == Constants.Any)
311                     {
312                         value = source;
313                     }
314                     if (storedLinkValue->Source == value ||
315                         storedLinkValue->Target == value)
316                     {
317                         return handler(GetLinkStruct(index));
318                     }
319                     return Constants.Continue;
320                 }
321             }
322             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪ поддерживаются.");
323         }
324 
325         /// <remarks>
326         /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
                ↪ в другом месте (но не в менеджере памяти, а в логике Links)
327         /// </remarks>
328         [MethodImpl(MethodImplOptions.AggressiveInlining)]
329         public id Update(IList<id> restrictions, IList<id> substitution)
```

```
330         {
331             var linkIndex = restrictions[Constants.IndexPart];
332             var link = GetLinkUnsafe(linkIndex);
333             // Будет корректно работать только в том случае, если пространство выделенной связи
                ↪  предварительно заполнено нулями
334             if (link->Source != Constants.Null)
335             {
336                 _sourcesTreeMethods.Detach(ref _header->FirstAsSource, linkIndex);
337             }
338             if (link->Target != Constants.Null)
339             {
340                 _targetsTreeMethods.Detach(ref _header->FirstAsTarget, linkIndex);
341             }
342 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
343             var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
344             var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
345             if (leftTreeSize != rightTreeSize)
346             {
347                 throw new Exception("One of the trees is broken.");
348             }
349 #endif
350             link->Source = substitution[Constants.SourcePart];
351             link->Target = substitution[Constants.TargetPart];
352             if (link->Source != Constants.Null)
353             {
354                 _sourcesTreeMethods.Attach(ref _header->FirstAsSource, linkIndex);
355             }
356             if (link->Target != Constants.Null)
357             {
358                 _targetsTreeMethods.Attach(ref _header->FirstAsTarget, linkIndex);
359             }
360 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
361             leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
362             rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
363             if (leftTreeSize != rightTreeSize)
364             {
365                 throw new Exception("One of the trees is broken.");
366             }
367 #endif
368             return linkIndex;
369         }
370 
371         [MethodImpl(MethodImplOptions.AggressiveInlining)]
372         public IList<id> GetLinkStruct(id linkIndex)
373         {
374             var link = GetLinkUnsafe(linkIndex);
375             return new UInt64Link(linkIndex, link->Source, link->Target);
376         }
377 
378         [MethodImpl(MethodImplOptions.AggressiveInlining)]
379         internal UInt64RawLink* GetLinkUnsafe(id linkIndex) => &_links[linkIndex];
380 
381         /// <remarks>
382         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
            ↪  пространство
383         /// </remarks>
384         public id Create(IList<id> restritions)
385         {
386             var freeLink = _header->FirstFreeLink;
387             if (freeLink != Constants.Null)
388             {
389                 _unusedLinksListMethods.Detach(freeLink);
390             }
391             else
392             {
393                 var maximumPossibleInnerReference =
                    ↪  Constants.PossibleInnerReferencesRange.Maximum;
394                 if (_header->AllocatedLinks > maximumPossibleInnerReference)
395                 {
396                     throw new LinksLimitReachedException<id>(maximumPossibleInnerReference);
397                 }
398                 if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
399                 {
400                     _memory.ReservedCapacity += _memoryReservationStep;
401                     SetPointers(_memory);
402                     _header->ReservedLinks = (id)(_memory.ReservedCapacity /
                        ↪  sizeof(UInt64RawLink));
403                 }
404                 _header->AllocatedLinks++;
```

```
405                 _memory.UsedCapacity += sizeof(UInt64RawLink);
406                 freeLink = _header->AllocatedLinks;
407             }
408             return freeLink;
409         }
410 
411         public void Delete(IList<id> restrictions)
412         {
413             var link = restrictions[Constants.IndexPart];
414             if (link < _header->AllocatedLinks)
415             {
416                 _unusedLinksListMethods.AttachAsFirst(link);
417             }
418             else if (link == _header->AllocatedLinks)
419             {
420                 _header->AllocatedLinks--;
421                 _memory.UsedCapacity -= sizeof(UInt64RawLink);
422                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪  пока не дойдём до первой существующей связи
423                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
424                 while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
425                 {
426                     _unusedLinksListMethods.Detach(_header->AllocatedLinks);
427                     _header->AllocatedLinks--;
428                     _memory.UsedCapacity -= sizeof(UInt64RawLink);
429                 }
430             }
431         }
432 
433         /// <remarks>
434         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
            ↪  адрес реально поменялся
435         ///
436         /// Указатель this.links может быть в том же месте,
437         /// так как 0-я связь не используется и имеет такой же размер как Header,
438         /// поэтому header размещается в том же месте, что и 0-я связь
439         /// </remarks>
440         private void SetPointers(IResizableDirectMemory memory)
441         {
442             if (memory == null)
443             {
444                 _header = null;
445                 _links = null;
446                 _unusedLinksListMethods = null;
447                 _targetsTreeMethods = null;
448                 _unusedLinksListMethods = null;
449             }
450             else
451             {
452                 _header = (UInt64LinksHeader*)(void*)memory.Pointer;
453                 _links = (UInt64RawLink*)(void*)memory.Pointer;
454                 _sourcesTreeMethods = new UInt64LinksSourcesAVLBalancedTreeMethods(this, _links,
                        ↪  _header);
455                 _targetsTreeMethods = new UInt64LinksTargetsAVLBalancedTreeMethods(this, _links,
                        ↪  _header);
456                 _unusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
457             }
458         }
459 
460         [MethodImpl(MethodImplOptions.AggressiveInlining)]
461         private bool Exists(id link) => link >= Constants.PossibleInnerReferencesRange.Minimum
            ↪  && link <= _header->AllocatedLinks && !IsUnusedLink(link);
462 
463         [MethodImpl(MethodImplOptions.AggressiveInlining)]
464         private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
465                                         || (_links[link].SizeAsSource == Constants.Null &&
                                            ↪  _links[link].Source != Constants.Null);
466 
467         #region Disposable
468 
469         protected override bool AllowMultipleDisposeCalls => true;
470 
471         protected override void Dispose(bool manual, bool wasDisposed)
472         {
473             if (!wasDisposed)
474             {
475                 SetPointers(null);
476                 _memory.DisposeIfPossible();
477             }
```

```
        }

        #endregion
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Collections.Generic;

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

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

        public override TLink Convert(IList<TLink> sequence)
        {
            var length = sequence.Count;
            if (length < 1)
```

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

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

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

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

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

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

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

        private struct HalfDoublet
        {
```

```
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;
98                  if (_doInitialFrequenciesIncrement)
99                  {
100                     data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
101                 }
102                 else
103                 {
104                     data = _doubletFrequenciesCache.GetFrequency(ref doublet);
105                     if (data == null)
```

```
                    {
                        throw new NotSupportedException("If you ask not to increment frequencies, it is expected that all frequencies for the sequence are prepared.");
                    }
                }
                copy[i - 1].Element = sequence[i - 1];
                copy[i - 1].DoubletData = data;
                UpdateMaxDoublet(ref doublet, data);
            }
            copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
            copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
            if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
            {
                var newLength = ReplaceDoublets(copy);
                sequence = new TLink[newLength];
                for (int i = 0; i < newLength; i++)
                {
                    sequence[i] = copy[i].Element;
                }
            }
            return sequence;
        }

        /// <remarks>
        /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
        /// </remarks>
        private int ReplaceDoublets(HalfDoublet[] copy)
        {
            var oldLength = copy.Length;
            var newLength = copy.Length;
            while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
            {
                var maxDoubletSource = _maxDoublet.Source;
                var maxDoubletTarget = _maxDoublet.Target;
                if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
                {
                    _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
                }
                var maxDoubletReplacementLink = _maxDoubletData.Link;
                oldLength--;
                var oldLengthMinusTwo = oldLength - 1;
                // Substitute all usages
                int w = 0, r = 0; // (r == read, w == write)
                for (; r < oldLength; r++)
                {
                    if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) && _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
                    {
                        if (r > 0)
                        {
                            var previous = copy[w - 1].Element;
                            copy[w - 1].DoubletData.DecrementFrequency();
                            copy[w - 1].DoubletData = _doubletFrequenciesCache.IncrementFrequency(previous, maxDoubletReplacementLink);
                        }
                        if (r < oldLengthMinusTwo)
                        {
                            var next = copy[r + 2].Element;
                            copy[r + 1].DoubletData.DecrementFrequency();
                            copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink, next);
                        }
                        copy[w++].Element = maxDoubletReplacementLink;
                        r++;
                        newLength--;
                    }
                    else
                    {
                        copy[w++] = copy[r];
                    }
                }
                if (w < newLength)
                {
                    copy[w] = copy[r];
                }
```

```
                oldLength = newLength;
                ResetMaxDoublet();
                UpdateMaxDoublet(copy, newLength);
            }
            return newLength;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
        {
            var frequency = data.Frequency;
            var maxFrequency = _maxDoubletData.Frequency;
            //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
            ↪  (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
            ↪  compression string data (and gives collisions quickly) */ _maxDoublet.Source +
            ↪  _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) */
            {
                _maxDoublet = doublet;
                _maxDoubletData = data;
            }
        }
    }
}
```

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

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

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

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

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

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

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

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

```
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;

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

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

```
                        }
                    }
                }
                length = w;
            }
            return links.GetOrCreate(sequence[0], sequence[1]);
        }

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

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

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

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

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

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

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

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

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

        public IList<TLink> Convert(IList<TLink> sequence)
        {
            var levels = new TLink[sequence.Count];
            levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
            for (var i = 1; i < sequence.Count - 1; i++)
            {
                var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
                var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
                levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
            }
            levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
            ↪  sequence[sequence.Count - 1]);
            return levels;
        }

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

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

```
using Platform.Interfaces;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
        ↪  ISequenceHeightProvider<TLink> heightProvider)
            : base(links)
        {
            _stack = stack;
            _heightProvider = heightProvider;
        }

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

```
            }
            return Links.GetOrCreate(left, right);
        }
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

```
                }
                return intermediateResult;
            }
        }

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

        public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
        {
            _groups = new HashSet<KeyValuePair<IList<TLink>,
            ↪  IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
            var count = _links.Count();
            _visited = new BitString((long)(Integer<TLink>)count + 1);
            _links.Each(link =>
            {
                var linkIndex = _links.GetIndex(link);
                var linkBitIndex = (long)(Integer<TLink>)linkIndex;
                if (!_visited.Get(linkBitIndex))
                {
                    var sequenceElements = new List<TLink>();
                    var filler = new ListFiller<TLink, TLink>(sequenceElements,
                    ↪  _sequences.Constants.Break);
                    _sequences.Each(filler.AddAllValuesAndReturnConstant, new
                    ↪  LinkAddress<TLink>(linkIndex));
                    if (sequenceElements.Count > 2)
                    {
                        WalkAll(sequenceElements);
                    }
                }
                return _links.Constants.Continue;
            });
            var resultList = _groups.ToList();
            var comparer = Default<ItemComparer>.Instance;
            resultList.Sort(comparer);
#if DEBUG
            foreach (var item in resultList)
            {
                PrintDuplicates(item);
            }
#endif
            return resultList;
        }

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

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

        private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
        {
            var duplicates = new List<TLink>();
            var readAsElement = new HashSet<TLink>();
            var restrictions = segment.ConvertToRestrictionsValues();
            restrictions[0] = _sequences.Constants.Any;
            _sequences.Each(sequence =>
            {
                var sequenceIndex = sequence[_sequences.Constants.IndexPart];
                duplicates.Add(sequenceIndex);
                readAsElement.Add(sequenceIndex);
                return _sequences.Constants.Continue;
            }, restrictions);
            if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
            {
                return new List<TLink>();
            }
```

```
            foreach (var duplicate in duplicates)
            {
                var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
                _visited.Set(duplicateBitIndex);
            }
            if (_sequences is Sequences sequencesExperiments)
            {
                var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((HashSet<ulong>)(object)readAsElement, (IList<ulong>)segment);
                foreach (var partiallyMatchedSequence in partiallyMatched)
                {
                    TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
                    duplicates.Add(sequenceIndex);
                }
            }
            duplicates.Sort();
            return duplicates;
        }

        private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
        {
            if (!(_links is ILinks<ulong> ulongLinks))
            {
                return;
            }
            var duplicatesKey = duplicatesItem.Key;
            var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
            Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
            var duplicatesList = duplicatesItem.Value;
            for (int i = 0; i < duplicatesList.Count; i++)
            {
                ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
                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));
                Console.WriteLine(formatedSequenceStructure);
                var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex, ulongLinks);
                Console.WriteLine(sequenceString);
            }
            Console.WriteLine();
        }
    }
}
```

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

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

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

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

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

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

```
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))
```

```
                        || ((_comparer.Compare(count, frequency) > 0) &&
                        ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency),
                        ↪  Integer<TLink>.One) > 0)))
                    {
                        throw new InvalidOperationException("Frequencies validation failed.");
                    }
                }
                //else
                //{
                //    if (value.Frequency > 0)
                //    {
                //        var frequency = value.Frequency;
                //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
                //        var count = _countLinkFrequency(linkIndex);

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

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

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

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

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

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

        public LinkFrequency() { }

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

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

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

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

```
using Platform.Interfaces;

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

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

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

```
using Platform.Interfaces;

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using Platform.Interfaces;

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

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

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

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

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

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

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

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

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

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

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

```
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
    {
        private readonly ILinks<TLink> _links;
        public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
        public TLink Count(TLink symbol) => new
        ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
```

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

        protected readonly ILinks<TLink> _links;
        protected readonly TLink _symbol;
        protected readonly HashSet<TLink> _visits;
        protected TLink _total;

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

```
using Platform.Interfaces;

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

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

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

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

```

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

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

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

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

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

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

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

        private readonly LinkFrequenciesCache<TLink> _cache;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Collections.Generic;

```

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

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

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

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

```
using System.Collections.Generic;

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

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

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

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

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

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

```
using System.Collections.Generic;

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

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

        private readonly ISynchronizedLinks<TLink> _links;

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

        public bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            var links = _links.Unsync;
            _links.SyncRoot.ExecuteReadOperation(() =>
            {
```

```
                while (--i >= 1 && (indexed =
                    !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                    sequence[i]), default))) { }
            });
            if (!indexed)
            {
                _links.SyncRoot.ExecuteWriteOperation(() =>
                {
                    for (; i >= 1; i--)
                    {
                        links.GetOrCreate(sequence[i - 1], sequence[i]);
                    }
                });
            }
            return indexed;
        }

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

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

```
using System.Collections.Generic;

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

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

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

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

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

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

namespace Platform.Data.Doublets.Sequences
{
    public class ListFiller<TElement, TReturnConstant>
    {
        protected readonly List<TElement> _list;
        protected readonly TReturnConstant _returnConstant;

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

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

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

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

```

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// Представляет коллекцию последовательностей связей.
    /// </summary>
    /// <remarks>
    /// Обязательно реализовать атомарность каждого публичного метода.
    ///
    /// TODO:
    ///
    /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
    /// через естественную группировку по unicode типам, все whitespace вместе, все символы вместе, все числа вместе и т.п.
    /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину графа)
    ///
    /// x*y - найти все связи между, в последовательностях любой формы, если не стоит ограничитель на то, что является последовательностью, а что нет,
    /// то находятся любые структуры связей, которые содержат эти элементы именно в таком порядке.
    ///
    /// Рост последовательности слева и справа.
    /// Поиск со звёздочкой.
    /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
    /// так же проблема может быть решена при реализации дистанционных триггеров.
    /// Нужны ли уникальные указатели вообще?
    /// Что если обращение к информации будет происходить через содержимое всегда?
    ///
    /// Писать тесты.
    ///
    ///
```

```
43      /// Можно убрать зависимость от конкретной реализации Links,
44      /// на зависимость от абстрактного элемента, который может быть представлен несколькими
        ↪ способами.
45      ///
46      /// Можно ли как-то сделать один общий интерфейс
47      ///
48      ///
49      /// Блокчейн и/или гит для распределённой записи транзакций.
50      ///
51      /// </remarks>
52      public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
        ↪ (после завершения реализации Sequences)
53      {
54          /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
            ↪ связей.</summary>
55          public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
56
57          public SequencesOptions<LinkIndex> Options { get; }
58          public SynchronizedLinks<LinkIndex> Links { get; }
59          private readonly ISynchronization _sync;
60
61          public LinksConstants<LinkIndex> Constants { get; }
62
63          public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
64          {
65              Links = links;
66              _sync = links.SyncRoot;
67              Options = options;
68              Options.ValidateOptions();
69              Options.InitOptions(Links);
70              Constants = Default<LinksConstants<LinkIndex>>.Instance;
71          }
72
73          public Sequences(SynchronizedLinks<LinkIndex> links)
74              : this(links, new SequencesOptions<LinkIndex>())
75          {
76          }
77
78          public bool IsSequence(LinkIndex sequence)
79          {
80              return _sync.ExecuteReadOperation(() =>
81              {
82                  if (Options.UseSequenceMarker)
83                  {
84                      return Options.MarkedSequenceMatcher.IsMatched(sequence);
85                  }
86                  return !Links.Unsync.IsPartialPoint(sequence);
87              });
88          }
89
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          private LinkIndex GetSequenceByElements(LinkIndex sequence)
92          {
93              if (Options.UseSequenceMarker)
94              {
95                  return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
96              }
97              return sequence;
98          }
99
100         private LinkIndex GetSequenceElements(LinkIndex sequence)
101         {
102             if (Options.UseSequenceMarker)
103             {
104                 var linkContents = new UInt64Link(Links.GetLink(sequence));
105                 if (linkContents.Source == Options.SequenceMarkerLink)
106                 {
107                     return linkContents.Target;
108                 }
109                 if (linkContents.Target == Options.SequenceMarkerLink)
110                 {
111                     return linkContents.Source;
112                 }
113             }
114             return sequence;
115         }
116
117         #region Count
118
```

```
119         public LinkIndex Count(IList<LinkIndex> restrictions)
120         {
121             if (restrictions.IsNullOrEmpty())
122             {
123                 return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
124             }
125             if (restrictions.Count == 1) // Первая связь это адрес
126             {
127                 var sequenceIndex = restrictions[0];
128                 if (sequenceIndex == Constants.Null)
129                 {
130                     return 0;
131                 }
132                 if (sequenceIndex == Constants.Any)
133                 {
134                     return Count(null);
135                 }
136                 if (Options.UseSequenceMarker)
137                 {
138                     return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
139                 }
140                 return Links.Exists(sequenceIndex) ? 1UL : 0;
141             }
142             throw new NotImplementedException();
143         }
144 
145         private LinkIndex CountUsages(params LinkIndex[] restrictions)
146         {
147             if (restrictions.Length == 0)
148             {
149                 return 0;
150             }
151             if (restrictions.Length == 1) // Первая связь это адрес
152             {
153                 if (restrictions[0] == Constants.Null)
154                 {
155                     return 0;
156                 }
157                 if (Options.UseSequenceMarker)
158                 {
159                     var elementsLink = GetSequenceElements(restrictions[0]);
160                     var sequenceLink = GetSequenceByElements(elementsLink);
161                     if (sequenceLink != Constants.Null)
162                     {
163                         return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
164                     }
165                     return Links.Count(elementsLink);
166                 }
167                 return Links.Count(restrictions[0]);
168             }
169             throw new NotImplementedException();
170         }
171 
172         #endregion
173 
174         #region Create
175 
176         public LinkIndex Create(IList<LinkIndex> restrictions)
177         {
178             return _sync.ExecuteWriteOperation(() =>
179             {
180                 if (restrictions.IsNullOrEmpty())
181                 {
182                     return Constants.Null;
183                 }
184                 Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
185                 return CreateCore(restrictions);
186             });
187         }
188 
189         private LinkIndex CreateCore(IList<LinkIndex> restrictions)
190         {
191             LinkIndex[] sequence = restrictions.ExtractValues();
192             if (Options.UseIndex)
193             {
194                 Options.Index.Add(sequence);
195             }
196             var sequenceRoot = default(LinkIndex);
197             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
```

```
198                 {
199                     var matches = Each(restrictions);
200                     if (matches.Count > 0)
201                     {
202                         sequenceRoot = matches[0];
203                     }
204                 }
205                 else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
206                 {
207                     return CompactCore(sequence);
208                 }
209                 if (sequenceRoot == default)
210                 {
211                     sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
212                 }
213                 if (Options.UseSequenceMarker)
214                 {
215                     Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
216                 }
217                 return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
218             }
219
220             #endregion
221
222             #region Each
223
224             public List<LinkIndex> Each(IList<LinkIndex> sequence)
225             {
226                 var results = new List<LinkIndex>();
227                 var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
228                 Each(filler.AddFirstAndReturnConstant, sequence);
229                 return results;
230             }
231
232             public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↪  restrictions)
233             {
234                 return _sync.ExecuteReadOperation(() =>
235                 {
236                     if (restrictions.IsNullOrEmpty())
237                     {
238                         return Constants.Continue;
239                     }
240                     Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
241                     if (restrictions.Count == 1)
242                     {
243                         var link = restrictions[0];
244                         var any = Constants.Any;
245                         if (link == any)
246                         {
247                             if (Options.UseSequenceMarker)
248                             {
249                                 return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
    ↪  Options.SequenceMarkerLink, any));
250                             }
251                             else
252                             {
253                                 return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
    ↪  any));
254                             }
255                         }
256                         var sequence =
    ↪  Options.Walker.Walk(link).ToArray().ConvertToRestrictionsValues();
257                         sequence[0] = link;
258                         return handler(sequence);
259                     }
260                     else if (restrictions.Count == 2)
261                     {
262                         throw new NotImplementedException();
263                     }
264                     else if (restrictions.Count == 3)
265                     {
266                         return Links.Unsync.Each(handler, restrictions);
267                     }
268                     else
269                     {
270                         var sequence = restrictions.ExtractValues();
271                         if (Options.UseIndex && !Options.Index.MightContain(sequence))
```

```
                {
                    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)
    {
        var upStep = stepFrom;
        var firstSource = Links.Unsync.GetTarget(upStep);
        while (firstSource != right && firstSource != upStep)
        {
            upStep = firstSource;
            firstSource = Links.Unsync.GetSource(upStep);
        }
        if (firstSource == right)
        {
            return handler(new LinkAddress<LinkIndex>(stepFrom));
```

```
339             }
340             return Constants.Continue;
341         }
342 
343         private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
            ↪ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
            ↪ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
            ↪ right));
344 
345         private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪ left, LinkIndex stepFrom)
346         {
347             var upStep = stepFrom;
348             var firstTarget = Links.Unsync.GetSource(upStep);
349             while (firstTarget != left && firstTarget != upStep)
350             {
351                 upStep = firstTarget;
352                 firstTarget = Links.Unsync.GetTarget(upStep);
353             }
354             if (firstTarget == left)
355             {
356                 return handler(new LinkAddress<LinkIndex>(stepFrom));
357             }
358             return Constants.Continue;
359         }
360 
361         #endregion
362 
363         #region Update
364 
365         public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
366         {
367             var sequence = restrictions.ExtractValues();
368             var newSequence = substitution.ExtractValues();
369 
370             if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
371             {
372                 return Constants.Null;
373             }
374             if (sequence.IsNullOrEmpty())
375             {
376                 return Create(substitution);
377             }
378             if (newSequence.IsNullOrEmpty())
379             {
380                 Delete(restrictions);
381                 return Constants.Null;
382             }
383             return _sync.ExecuteWriteOperation(() =>
384             {
385                 Links.EnsureEachLinkIsAnyOrExists(sequence);
386                 Links.EnsureEachLinkExists(newSequence);
387                 return UpdateCore(sequence, newSequence);
388             });
389         }
390 
391         private LinkIndex UpdateCore(LinkIndex[] sequence, LinkIndex[] newSequence)
392         {
393             LinkIndex bestVariant;
394             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
                ↪ !sequence.EqualTo(newSequence))
395             {
396                 bestVariant = CompactCore(newSequence);
397             }
398             else
399             {
400                 bestVariant = CreateCore(newSequence);
401             }
402             // TODO: Check all options only ones before loop execution
403             // Возможно нужно две версии Each, возвращающий фактические последовательности и с
                ↪ маркером,
404             // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
                ↪ можно получить имея только фактические последовательности.
405             foreach (var variant in Each(sequence))
406             {
407                 if (variant != bestVariant)
408                 {
409                     UpdateOneCore(variant, bestVariant);
410                 }
```

```
            }
            return bestVariant;
        }

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

        #endregion

        #region Delete

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

        private void DeleteOneCore(LinkIndex link)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(link);
                var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                if (Options.UseCascadeDelete || CountUsages(link) == 0)
                {
                    if (sequenceLink != Constants.Null)
```

```
489                 {
490                     Links.Unsync.Delete(sequenceLink);
491                 }
492                 Links.Unsync.Delete(link);
493             }
494             ClearGarbage(sequenceElementsContents.Source);
495             ClearGarbage(sequenceElementsContents.Target);
496         }
497         else
498         {
499             if (Options.UseSequenceMarker)
500             {
501                 var sequenceElements = GetSequenceElements(link);
502                 var sequenceLink = GetSequenceByElements(sequenceElements);
503                 if (Options.UseCascadeDelete || CountUsages(link) == 0)
504                 {
505                     if (sequenceLink != Constants.Null)
506                     {
507                         Links.Unsync.Delete(sequenceLink);
508                     }
509                     Links.Unsync.Delete(link);
510                 }
511             }
512             else
513             {
514                 if (Options.UseCascadeDelete || CountUsages(link) == 0)
515                 {
516                     Links.Unsync.Delete(link);
517                 }
518             }
519         }
520     }
521
522     #endregion
523
524     #region Compactification
525
526     /// <remarks>
527     /// bestVariant можно выбирать по максимальному числу использований,
528     /// но балансированный позволяет гарантировать уникальность (если есть возможность,
529     /// гарантировать его использование в других местах).
530     ///
531     /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
532     /// </remarks>
533     public LinkIndex Compact(params LinkIndex[] sequence)
534     {
535         return _sync.ExecuteWriteOperation(() =>
536         {
537             if (sequence.IsNullOrEmpty())
538             {
539                 return Constants.Null;
540             }
541             Links.EnsureEachLinkExists(sequence);
542             return CompactCore(sequence);
543         });
544     }
545
546     [MethodImpl(MethodImplOptions.AggressiveInlining)]
547     private LinkIndex CompactCore(params LinkIndex[] sequence) => UpdateCore(sequence,
        ↪  sequence);
548
549     #endregion
550
551     #region Garbage Collection
552
553     /// <remarks>
554     /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
        ↪  определить извне или в унаследованном классе
555     /// </remarks>
556     [MethodImpl(MethodImplOptions.AggressiveInlining)]
557     private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
        ↪  !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;
558
559     private void ClearGarbage(LinkIndex link)
560     {
561         if (IsGarbage(link))
562         {
563             var contents = new UInt64Link(Links.GetLink(link));
564             Links.Unsync.Delete(link);
```

```
565                 ClearGarbage(contents.Source);
566                 ClearGarbage(contents.Target);
567             }
568         }
569 
570         #endregion
571 
572         #region Walkers
573 
574         public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
575         {
576             return _sync.ExecuteReadOperation(() =>
577             {
578                 var links = Links.Unsync;
579                 foreach (var part in Options.Walker.Walk(sequence))
580                 {
581                     if (!handler(part))
582                     {
583                         return false;
584                     }
585                 }
586                 return true;
587             });
588         }
589 
590         public class Matcher : RightSequenceWalker<LinkIndex>
591         {
592             private readonly Sequences _sequences;
593             private readonly IList<LinkIndex> _patternSequence;
594             private readonly HashSet<LinkIndex> _linksInSequence;
595             private readonly HashSet<LinkIndex> _results;
596             private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
597             private readonly HashSet<LinkIndex> _readAsElements;
598             private int _filterPosition;
599 
600             public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
                ↪  HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
                ↪  HashSet<LinkIndex> readAsElements = null)
601                 : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
602             {
603                 _sequences = sequences;
604                 _patternSequence = patternSequence;
605                 _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪  Links.Constants.Any && x != ZeroOrMany));
606                 _results = results;
607                 _stopableHandler = stopableHandler;
608                 _readAsElements = readAsElements;
609             }
610 
611             protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
                ↪  (_readAsElements != null && _readAsElements.Contains(link)) ||
                ↪  _linksInSequence.Contains(link);
612 
613             public bool FullMatch(LinkIndex sequenceToMatch)
614             {
615                 _filterPosition = 0;
616                 foreach (var part in Walk(sequenceToMatch))
617                 {
618                     if (!FullMatchCore(part))
619                     {
620                         break;
621                     }
622                 }
623                 return _filterPosition == _patternSequence.Count;
624             }
625 
626             private bool FullMatchCore(LinkIndex element)
627             {
628                 if (_filterPosition == _patternSequence.Count)
629                 {
630                     _filterPosition = -2; // Длиннее чем нужно
631                     return false;
632                 }
633                 if (_patternSequence[_filterPosition] != Links.Constants.Any
634                  && element != _patternSequence[_filterPosition])
635                 {
636                     _filterPosition = -1;
637                     return false; // Начинается/Продолжается иначе
638                 }
639                 _filterPosition++;
```

```
            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)
            {
                if (element == _patternSequence[0])
                {
                    _filterPosition = 0;
                }
            }
            return true; // Ищем дальше
        }

        public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
        {
```

```
                if (PartialMatch(sequenceToMatch))
                {
                    _results.Add(sequenceToMatch);
                }
            }

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

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

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

        #endregion
    }
}
```

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

```
using System;
using LinkIndex = System.UInt64;
using System.Collections.Generic;
using Stack = System.Collections.Generic.Stack<ulong>;
using System.Linq;
using System.Text;
using Platform.Collections;
using Platform.Data.Exceptions;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Collections.Stacks;

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

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

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

```
35                  if (sequence.Length == 1)
36                  {
37                      return sequence;
38                  }
39                  return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
40              });
41          }
42
43          private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
44          {
45  #if DEBUG
46              if ((stopAt - startAt) < 0)
47              {
48                  throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
                    ↪  меньше или равен stopAt");
49              }
50  #endif
51              if ((stopAt - startAt) == 0)
52              {
53                  return new[] { sequence[startAt] };
54              }
55              if ((stopAt - startAt) == 1)
56              {
57                  return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt], sequence[stopAt])
                    ↪  };
58              }
59              var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
60              var last = 0;
61              for (var splitter = startAt; splitter < stopAt; splitter++)
62              {
63                  var left = CreateAllVariants2Core(sequence, startAt, splitter);
64                  var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
65                  for (var i = 0; i < left.Length; i++)
66                  {
67                      for (var j = 0; j < right.Length; j++)
68                      {
69                          var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
70                          if (variant == Constants.Null)
71                          {
72                              throw new NotImplementedException("Creation cancellation is not
                                ↪  implemented.");
73                          }
74                          variants[last++] = variant;
75                      }
76                  }
77              }
78              return variants;
79          }
80
81          public List<ulong> CreateAllVariants1(params ulong[] sequence)
82          {
83              return _sync.ExecuteWriteOperation(() =>
84              {
85                  if (sequence.IsNullOrEmpty())
86                  {
87                      return new List<ulong>();
88                  }
89                  Links.Unsync.EnsureEachLinkExists(sequence);
90                  if (sequence.Length == 1)
91                  {
92                      return new List<ulong> { sequence[0] };
93                  }
94                  var results = new
                    ↪  List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
95                  return CreateAllVariants1Core(sequence, results);
96              });
97          }
98
99          private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
100         {
101             if (sequence.Length == 2)
102             {
103                 var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
104                 if (link == Constants.Null)
105                 {
106                     throw new NotImplementedException("Creation cancellation is not
                        ↪  implemented.");
107                 }
```

```
108                 results.Add(link);
109                 return results;
110             }
111             var innerSequenceLength = sequence.Length - 1;
112             var innerSequence = new ulong[innerSequenceLength];
113             for (var li = 0; li < innerSequenceLength; li++)
114             {
115                 var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
116                 if (link == Constants.Null)
117                 {
118                     throw new NotImplementedException("Creation cancellation is not
                        ↪ implemented.");
119                 }
120                 for (var isi = 0; isi < li; isi++)
121                 {
122                     innerSequence[isi] = sequence[isi];
123                 }
124                 innerSequence[li] = link;
125                 for (var isi = li + 1; isi < innerSequenceLength; isi++)
126                 {
127                     innerSequence[isi] = sequence[isi + 1];
128                 }
129                 CreateAllVariants1Core(innerSequence, results);
130             }
131             return results;
132         }
133 
134         #endregion
135 
136         public HashSet<ulong> Each1(params ulong[] sequence)
137         {
138             var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
139             Each1(link =>
140             {
141                 if (!visitedLinks.Contains(link))
142                 {
143                     visitedLinks.Add(link); // изучить почему случаются повторы
144                 }
145                 return true;
146             }, sequence);
147             return visitedLinks;
148         }
149 
150         private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
151         {
152             if (sequence.Length == 2)
153             {
154                 Links.Unsync.Each(sequence[0], sequence[1], handler);
155             }
156             else
157             {
158                 var innerSequenceLength = sequence.Length - 1;
159                 for (var li = 0; li < innerSequenceLength; li++)
160                 {
161                     var left = sequence[li];
162                     var right = sequence[li + 1];
163                     if (left == 0 && right == 0)
164                     {
165                         continue;
166                     }
167                     var linkIndex = li;
168                     ulong[] innerSequence = null;
169                     Links.Unsync.Each(doublet =>
170                     {
171                         if (innerSequence == null)
172                         {
173                             innerSequence = new ulong[innerSequenceLength];
174                             for (var isi = 0; isi < linkIndex; isi++)
175                             {
176                                 innerSequence[isi] = sequence[isi];
177                             }
178                             for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
179                             {
180                                 innerSequence[isi] = sequence[isi + 1];
181                             }
182                         }
183                         innerSequence[linkIndex] = doublet[Constants.IndexPart];
184                         Each1(handler, innerSequence);
185                         return Constants.Continue;
```

```
                }, Constants.Any, left, right);
            }
        }
    }

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

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

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

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

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

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

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

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

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

```
341                 upStep = firstTarget;
342                 firstTarget = Links.Unsync.GetTarget(upStep);
343             }
344             if (firstTarget == left)
345             {
346                 handler(new LinkAddress<LinkIndex>(stepFrom));
347             }
348         }
349 
350         private bool StartsWith(ulong sequence, ulong link)
351         {
352             var upStep = sequence;
353             var firstSource = Links.Unsync.GetSource(upStep);
354             while (firstSource != link && firstSource != upStep)
355             {
356                 upStep = firstSource;
357                 firstSource = Links.Unsync.GetSource(upStep);
358             }
359             return firstSource == link;
360         }
361 
362         private bool EndsWith(ulong sequence, ulong link)
363         {
364             var upStep = sequence;
365             var lastTarget = Links.Unsync.GetTarget(upStep);
366             while (lastTarget != link && lastTarget != upStep)
367             {
368                 upStep = lastTarget;
369                 lastTarget = Links.Unsync.GetTarget(upStep);
370             }
371             return lastTarget == link;
372         }
373 
374         public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
375         {
376             return _sync.ExecuteReadOperation(() =>
377             {
378                 var results = new List<ulong>();
379                 if (sequence.Length > 0)
380                 {
381                     Links.EnsureEachLinkExists(sequence);
382                     var firstElement = sequence[0];
383                     if (sequence.Length == 1)
384                     {
385                         results.Add(firstElement);
386                         return results;
387                     }
388                     if (sequence.Length == 2)
389                     {
390                         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
391                         if (doublet != Constants.Null)
392                         {
393                             results.Add(doublet);
394                         }
395                         return results;
396                     }
397                     var linksInSequence = new HashSet<ulong>(sequence);
398                     void handler(IList<LinkIndex> result)
399                     {
400                         var resultIndex = result[Links.Constants.IndexPart];
401                         var filterPosition = 0;
402                         StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
                            ↪  Links.Unsync.GetTarget,
403                             x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                                ↪  x =>
404                             {
405                                 if (filterPosition == sequence.Length)
406                                 {
407                                     filterPosition = -2; // Длиннее чем нужно
408                                     return false;
409                                 }
410                                 if (x != sequence[filterPosition])
411                                 {
412                                     filterPosition = -1;
413                                     return false; // Начинается иначе
414                                 }
415                                 filterPosition++;
416 
417                                 return true;
```

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

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

    public const int MaxSequenceFormatSize = 200;

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

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

```
489
490         private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
            ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
            ↪ LinkIndex[] knownElements)
491         {
492             var linksInSequence = new HashSet<ulong>(knownElements);
493             //var entered = new HashSet<ulong>();
494             var sb = new StringBuilder();
495             sb.Append('{');
496             if (links.Exists(sequenceLink))
497             {
498                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
499                     x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
                        ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
500                     {
501                         if (insertComma && sb.Length > 1)
502                         {
503                             sb.Append(',');
504                         }
505                         //if (entered.Contains(element))
506                         //{
507                         //    sb.Append('{');
508                         //    elementToString(sb, element);
509                         //    sb.Append('}');
510                         //}
511                         //else
512                         elementToString(sb, element);
513                         if (sb.Length < MaxSequenceFormatSize)
514                         {
515                             return true;
516                         }
517                         sb.Append(insertComma ? ", ..." : "...");
518                         return false;
519                     });
520             }
521             sb.Append('}');
522             return sb.ToString();
523         }
524
525         public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
            ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
            ↪ knownElements);
526
527         public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
            ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
            ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
            ↪ sequenceLink, elementToString, insertComma, knownElements));
528
529         private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
            ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
            ↪ LinkIndex[] knownElements)
530         {
531             var linksInSequence = new HashSet<ulong>(knownElements);
532             var entered = new HashSet<ulong>();
533             var sb = new StringBuilder();
534             sb.Append('{');
535             if (links.Exists(sequenceLink))
536             {
537                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
538                     x => linksInSequence.Contains(x) || links.IsFullPoint(x),
                        ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
539                     {
540                         if (insertComma && sb.Length > 1)
541                         {
542                             sb.Append(',');
543                         }
544                         if (entered.Contains(element))
545                         {
546                             sb.Append('{');
547                             elementToString(sb, element);
548                             sb.Append('}');
549                         }
550                         else
551                         {
552                             elementToString(sb, element);
553                         }
554                         if (sb.Length < MaxSequenceFormatSize)
```

```
                    {
                        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>();
        });
    }

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

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

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

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

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

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

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

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

    //            firstResults.IntersectWith(lastResults);

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

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

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

    public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            if (sequence.Length > 0)
            {
                Links.EnsureEachLinkIsAnyOrExists(sequence);
                var firstResults = new HashSet<ulong>();
                var lastResults = new HashSet<ulong>();
                var first = sequence.First(x => x != Constants.Any);
                var last = sequence.Last(x => x != Constants.Any);
                AllUsagesCore(first, firstResults);
```

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

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

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

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

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

```
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
843                     //////            return new List<ulong>();
844                     //////    }
845                     //////}
846                     //////if (matches.Count > 0)
847                     //////{
848                     //////    var usages = new HashSet<ulong>();
849                     //////    for (int i = 0; i < sequence.Length; i++)
850                     //////    {
851                     //////        AllUsagesCore(sequence[i], usages);
852                     //////    }
853                     //////    //for (int i = 0; i < matches[0].Count; i++)
854                     //////    //    AllUsagesCore(matches[0][i], usages);
855                     //////    //usages.UnionWith(matches[0]);
```

```
                    //////    return usages.ToList();
                    //////}
                    var firstLinkUsages = new HashSet<ulong>();
                    AllUsagesCore(sequence[0], firstLinkUsages);
                    firstLinkUsages.Add(sequence[0]);
                    //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() { sequence[0] }; // or all sequences, containing this element?
                    //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages, 1).ToList();
                    var results = new HashSet<ulong>();
                    foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages, 1))
                    {
                        AllUsagesCore(match, results);
                    }
                    return results.ToList();
                }
                return new List<ulong>();
            });
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        private bool CalculateCore(ulong link)
        {
            // TODO: Проработать защиту от зацикливания
            // Основано на SequenceWalker.WalkLeft
            Func<ulong, ulong> getSource = _links.Unsync.GetSource;
            Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
            Func<ulong, bool> isElement = IsElement;
            void visitLeaf(ulong parent)
            {
                if (link != parent)
                {
                    _totals[parent]++;
                }
            }
            void visitNode(ulong parent)
            {
                if (link != parent)
                {
                    _totals[parent]++;
                }
            }
            var stack = new Stack();
            var element = link;
            if (isElement(element))
            {
                visitLeaf(element);
            }
            else
            {
                while (true)
                {
                    if (isElement(element))
                    {
                        if (stack.Count == 0)
                        {
                            break;
                        }
                        element = stack.Pop();
                        var source = getSource(element);
                        var target = getTarget(element);
                        // Обработка элемента
                        if (isElement(target))
                        {
                            visitLeaf(target);
```

```
                        }
                        if (isElement(source))
                        {
                            visitLeaf(source);
                        }
                        element = source;
                    }
                    else
                    {
                        stack.Push(element);
                        visitNode(element);
                        element = getTarget(element);
                    }
                }
            }
            _totals[link]++;
            return true;
        }
    }

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

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

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

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

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

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

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

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

        public bool Collect(ulong link)
        {
            if (_usages.Add((long)link))
```

```
            {
                _links.Each(link, _links.Constants.Any, Collect);
                _links.Each(_links.Constants.Any, link, Collect);
            }
            return true;
        }
    }

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

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

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

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

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

    private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
    ↪ HashSet<ulong> previousMatchings, long startAt)
    {
        if (startAt >= sequence.Length) // ?
        {
            return previousMatchings;
        }
        var secondLinkUsages = new HashSet<ulong>();
        AllUsagesCore(sequence[startAt], secondLinkUsages);
        secondLinkUsages.Add(sequence[startAt]);
        var matchings = new HashSet<ulong>();
```

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

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

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

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

```
1303            // Находит связи между всеми указанными связями в любом порядке.
1304            // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
                ↪  несколько раз в последовательности)
1305            public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1306            {
1307                return _sync.ExecuteReadOperation(() =>
1308                {
1309                    var results = new HashSet<ulong>();
1310                    if (linksToConnect.Length > 0)
1311                    {
1312                        Links.EnsureEachLinkExists(linksToConnect);
1313                        AllUsagesCore(linksToConnect[0], results);
1314                        for (var i = 1; i < linksToConnect.Length; i++)
1315                        {
1316                            var next = new HashSet<ulong>();
1317                            AllUsagesCore(linksToConnect[i], next);
1318                            results.IntersectWith(next);
1319                        }
1320                    }
1321                    return results;
1322                });
1323            }
1324
1325            public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1326            {
1327                return _sync.ExecuteReadOperation(() =>
1328                {
1329                    var results = new HashSet<ulong>();
1330                    if (linksToConnect.Length > 0)
1331                    {
1332                        Links.EnsureEachLinkExists(linksToConnect);
1333                        var collector1 = new AllUsagesCollector(Links.Unsync, results);
1334                        collector1.Collect(linksToConnect[0]);
1335                        var next = new HashSet<ulong>();
1336                        for (var i = 1; i < linksToConnect.Length; i++)
1337                        {
1338                            var collector = new AllUsagesCollector(Links.Unsync, next);
1339                            collector.Collect(linksToConnect[i]);
1340                            results.IntersectWith(next);
1341                            next.Clear();
1342                        }
1343                    }
1344                    return results;
1345                });
1346            }
1347
1348            public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1349            {
1350                return _sync.ExecuteReadOperation(() =>
1351                {
1352                    var results = new HashSet<ulong>();
1353                    if (linksToConnect.Length > 0)
1354                    {
1355                        Links.EnsureEachLinkExists(linksToConnect);
1356                        var collector1 = new AllUsagesCollector(Links, results);
1357                        collector1.Collect(linksToConnect[0]);
1358                        //AllUsagesCore(linksToConnect[0], results);
1359                        for (var i = 1; i < linksToConnect.Length; i++)
1360                        {
1361                            var next = new HashSet<ulong>();
1362                            var collector = new AllUsagesIntersectingCollector(Links, results, next);
1363                            collector.Collect(linksToConnect[i]);
1364                            //AllUsagesCore(linksToConnect[i], next);
1365                            //results.IntersectWith(next);
1366                            results = next;
1367                        }
1368                    }
1369                    return results;
1370                });
1371            }
1372
1373            public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1374            {
1375                return _sync.ExecuteReadOperation(() =>
1376                {
1377                    var results = new BitString((long)Links.Unsync.Count() + 1); // new
                        ↪  BitArray((int)_links.Total + 1);
1378                    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;
            }
            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>();
```

```
1456
1457        public void Prediction()
1458        {
1459            //_links
1460            //sequences
1461        }
1462
1463        #region From Triplets
1464
1465        //public static void DeleteSequence(Link sequence)
1466        //{
1467        //}
1468
1469        public List<ulong> CollectMatchingSequences(ulong[] links)
1470        {
1471            if (links.Length == 1)
1472            {
1473                throw new Exception("Подпоследовательности с одним элементом не
                    ↪  поддерживаются.");
1474            }
1475            var leftBound = 0;
1476            var rightBound = links.Length - 1;
1477            var left = links[leftBound++];
1478            var right = links[rightBound--];
1479            var results = new List<ulong>();
1480            CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1481            return results;
1482        }
1483
1484        private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
        ↪  middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1485        {
1486            var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1487            var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1488            if (leftLinkTotalReferers <= rightLinkTotalReferers)
1489            {
1490                var nextLeftLink = middleLinks[leftBound];
1491                var elements = GetRightElements(leftLink, nextLeftLink);
1492                if (leftBound <= rightBound)
1493                {
1494                    for (var i = elements.Length - 1; i >= 0; i--)
1495                    {
1496                        var element = elements[i];
1497                        if (element != 0)
1498                        {
1499                            CollectMatchingSequences(element, leftBound + 1, middleLinks,
                                ↪  rightLink, rightBound, ref results);
1500                        }
1501                    }
1502                }
1503                else
1504                {
1505                    for (var i = elements.Length - 1; i >= 0; i--)
1506                    {
1507                        var element = elements[i];
1508                        if (element != 0)
1509                        {
1510                            results.Add(element);
1511                        }
1512                    }
1513                }
1514            }
1515            else
1516            {
1517                var nextRightLink = middleLinks[rightBound];
1518                var elements = GetLeftElements(rightLink, nextRightLink);
1519                if (leftBound <= rightBound)
1520                {
1521                    for (var i = elements.Length - 1; i >= 0; i--)
1522                    {
1523                        var element = elements[i];
1524                        if (element != 0)
1525                        {
1526                            CollectMatchingSequences(leftLink, leftBound, middleLinks,
                                ↪  elements[i], rightBound - 1, ref results);
1527                        }
1528                    }
1529                }
```

```
                else
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != 0)
                        {
                            results.Add(element);
                        }
                    }
                }
            }
        }

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

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

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

```
                }
                return true;
            });
            if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
            {
                result[4] = leftLink;
            }
            return result;
        }

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

        #endregion

        #region Walkers

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

            #region Pattern Match

            enum PatternBlockType
            {
                Undefined,
                Gap,
                Elements
            }

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

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

            #endregion

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

```
1686                 _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                     ↪  _sequences.Constants.Any && x != ZeroOrMany));
1687                 _results = results;
1688                 _pattern = CreateDetailedPattern();
1689             }
1690 
1691             protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
                     ↪  base.IsElement(link);
1692 
1693             public bool PatternMatch(LinkIndex sequenceToMatch)
1694             {
1695                 _patternPosition = 0;
1696                 _sequencePosition = 0;
1697                 foreach (var part in Walk(sequenceToMatch))
1698                 {
1699                     if (!PatternMatchCore(part))
1700                     {
1701                         break;
1702                     }
1703                 }
1704                 return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
                     ↪  - 1 && _pattern[_patternPosition].Start == 0);
1705             }
1706 
1707             private List<PatternBlock> CreateDetailedPattern()
1708             {
1709                 var pattern = new List<PatternBlock>();
1710                 var patternBlock = new PatternBlock();
1711                 for (var i = 0; i < _patternSequence.Length; i++)
1712                 {
1713                     if (patternBlock.Type == PatternBlockType.Undefined)
1714                     {
1715                         if (_patternSequence[i] == _sequences.Constants.Any)
1716                         {
1717                             patternBlock.Type = PatternBlockType.Gap;
1718                             patternBlock.Start = 1;
1719                             patternBlock.Stop = 1;
1720                         }
1721                         else if (_patternSequence[i] == ZeroOrMany)
1722                         {
1723                             patternBlock.Type = PatternBlockType.Gap;
1724                             patternBlock.Start = 0;
1725                             patternBlock.Stop = long.MaxValue;
1726                         }
1727                         else
1728                         {
1729                             patternBlock.Type = PatternBlockType.Elements;
1730                             patternBlock.Start = i;
1731                             patternBlock.Stop = i;
1732                         }
1733                     }
1734                     else if (patternBlock.Type == PatternBlockType.Elements)
1735                     {
1736                         if (_patternSequence[i] == _sequences.Constants.Any)
1737                         {
1738                             pattern.Add(patternBlock);
1739                             patternBlock = new PatternBlock
1740                             {
1741                                 Type = PatternBlockType.Gap,
1742                                 Start = 1,
1743                                 Stop = 1
1744                             };
1745                         }
1746                         else if (_patternSequence[i] == ZeroOrMany)
1747                         {
1748                             pattern.Add(patternBlock);
1749                             patternBlock = new PatternBlock
1750                             {
1751                                 Type = PatternBlockType.Gap,
1752                                 Start = 0,
1753                                 Stop = long.MaxValue
1754                             };
1755                         }
1756                         else
1757                         {
1758                             patternBlock.Stop = i;
1759                         }
1760                     }
1761                     else // patternBlock.Type == PatternBlockType.Gap
1762                     {
```

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

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

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

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

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

```
1841                //    }
1842
1843                //    if (maximumGap < mininumGap)
1844                //        maximumGap = mininumGap;
1845                //}
1846
1847                private bool PatternMatchCore(LinkIndex element)
1848                {
1849                    if (_patternPosition >= _pattern.Count)
1850                    {
1851                        _patternPosition = -2;
1852                        return false;
1853                    }
1854                    var currentPatternBlock = _pattern[_patternPosition];
1855                    if (currentPatternBlock.Type == PatternBlockType.Gap)
1856                    {
1857                        //var currentMatchingBlockLength = (_sequencePosition -
                            ↪  _lastMatchedBlockPosition);
1858                        if (_sequencePosition < currentPatternBlock.Start)
1859                        {
1860                            _sequencePosition++;
1861                            return true; // Двигаемся дальше
1862                        }
1863                        // Это последний блок
1864                        if (_pattern.Count == _patternPosition + 1)
1865                        {
1866                            _patternPosition++;
1867                            _sequencePosition = 0;
1868                            return false; // Полное соответствие
1869                        }
1870                        else
1871                        {
1872                            if (_sequencePosition > currentPatternBlock.Stop)
1873                            {
1874                                return false; // Соответствие невозможно
1875                            }
1876                            var nextPatternBlock = _pattern[_patternPosition + 1];
1877                            if (_patternSequence[nextPatternBlock.Start] == element)
1878                            {
1879                                if (nextPatternBlock.Start < nextPatternBlock.Stop)
1880                                {
1881                                    _patternPosition++;
1882                                    _sequencePosition = 1;
1883                                }
1884                                else
1885                                {
1886                                    _patternPosition += 2;
1887                                    _sequencePosition = 0;
1888                                }
1889                            }
1890                        }
1891                    }
1892                    else // currentPatternBlock.Type == PatternBlockType.Elements
1893                    {
1894                        var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1895                        if (_patternSequence[patternElementPosition] != element)
1896                        {
1897                            return false; // Соответствие невозможно
1898                        }
1899                        if (patternElementPosition == currentPatternBlock.Stop)
1900                        {
1901                            _patternPosition++;
1902                            _sequencePosition = 0;
1903                        }
1904                        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                    ////////
```

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

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

        #endregion
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

        public void InitOptions(ISynchronizedLinks<TLink> links)
        {
            if (UseSequenceMarker)
            {
                if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
                {
                    SequenceMarkerLink = links.CreatePoint();
                }
                else
                {
                    if (!links.Exists(SequenceMarkerLink))
                    {
                        var link = links.CreatePoint();
                        if (!_equalityComparer.Equals(link, SequenceMarkerLink))
                        {
                            throw new InvalidOperationException("Cannot recreate sequence marker link.");
                        }
                    }
                }
                if (MarkedSequenceMatcher == null)
                {
                    MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links, SequenceMarkerLink);
                }
            }
            var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
            if (UseCompression)
            {
                if (LinksToSequenceConverter == null)
                {
                    ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
                    if (UseSequenceMarker)
                    {
```

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

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

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

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

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

namespace Platform.Data.Doublets.Sequences
{
    public class SetFiller<TElement, TReturnConstant>
    {
        protected readonly ISet<TElement> _set;
        protected readonly TReturnConstant _returnConstant;

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

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

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

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

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

```
            return true;
        }

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

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

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

```
using System.Collections.Generic;

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

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

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

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

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

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

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

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

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

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

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

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

```
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)
```

```
            {
                return array;
            }
            else
            {
                return CopyFilledElements(array, filledElementsCount);
            }
        }

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

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

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

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

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

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

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

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

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

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

```
                }
            }
        }
    }
}
```

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);

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

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

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

```
    }
}
```

./Platform.Data.Doublets/Stacks/Stack.cs
```
using System.Collections.Generic;
using Platform.Collections.Stacks;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets
{
    /// <remarks>
    /// TODO: Autogeneration of synchronized wrapper (decorator).
```

```
12      /// TODO: Try to unfold code of each method using IL generation for performance improvements.
13      /// TODO: Or even to unfold multiple layers of implementations.
14      /// </remarks>
15      public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
16      {
17          public LinksConstants<TLinkAddress> Constants { get; }
18          public ISynchronization SyncRoot { get; }
19          public ILinks<TLinkAddress> Sync { get; }
20          public ILinks<TLinkAddress> Unsync { get; }
21
22          public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
            ↪  ReaderWriterLockSynchronization(), links) { }
23
24          public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
25          {
26              SyncRoot = synchronization;
27              Sync = this;
28              Unsync = links;
29              Constants = links.Constants;
30          }
31
32          public TLinkAddress Count(IList<TLinkAddress> restriction) =>
            ↪  SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
33          public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
            ↪  IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
            ↪  restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
34          public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
            ↪  SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
35          public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
            ↪  substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
            ↪  Unsync.Update);
36          public void Delete(IList<TLinkAddress> restrictions) =>
            ↪  SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
37
38          //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
            ↪  IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
39          //{
40          //    if (restriction != null && substitution != null &&
            ↪  !substitution.EqualTo(restriction))
41          //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
            ↪  substitution, substitutedHandler, Unsync.Trigger);
42
43          //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
            ↪  substitutedHandler, Unsync.Trigger);
44          //}
45      }
46  }
```

./Platform.Data.Doublets/UInt64Link.cs

```
1   using System;
2   using System.Collections;
3   using System.Collections.Generic;
4   using Platform.Exceptions;
5   using Platform.Ranges;
6   using Platform.Singletons;
7   using Platform.Collections.Lists;
8
9   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11  namespace Platform.Data.Doublets
12  {
13      /// <summary>
14      /// Структура описывающая уникальную связь.
15      /// </summary>
16      public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
17      {
18          private static readonly LinksConstants<ulong> _constants =
            ↪  Default<LinksConstants<ulong>>.Instance;
19
20          private const int Length = 3;
21
22          public readonly ulong Index;
23          public readonly ulong Source;
24          public readonly ulong Target;
25
26          public static readonly UInt64Link Null = new UInt64Link();
27
28          public UInt64Link(params ulong[] values)
29          {
```

```
30              Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
                ↪  _constants.Null;
31              Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
                ↪  _constants.Null;
32              Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
                ↪  _constants.Null;
33          }
34
35          public UInt64Link(IList<ulong> values)
36          {
37              Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
                ↪  _constants.Null;
38              Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
                ↪  _constants.Null;
39              Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
                ↪  _constants.Null;
40          }
41
42          public UInt64Link(ulong index, ulong source, ulong target)
43          {
44              Index = index;
45              Source = source;
46              Target = target;
47          }
48
49          public UInt64Link(ulong source, ulong target)
50              : this(_constants.Null, source, target)
51          {
52              Source = source;
53              Target = target;
54          }
55
56          public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
            ↪  target);
57
58          public override int GetHashCode() => (Index, Source, Target).GetHashCode();
59
60          public bool IsNull() => Index == _constants.Null
61                                  && Source == _constants.Null
62                                  && Target == _constants.Null;
63
64          public override bool Equals(object other) => other is UInt64Link &&
            ↪  Equals((UInt64Link)other);
65
66          public bool Equals(UInt64Link other) => Index == other.Index
67                                                  && Source == other.Source
68                                                  && Target == other.Target;
69
70          public static string ToString(ulong index, ulong source, ulong target) => $"({index}:
            ↪  {source}->{target})";
71
72          public static string ToString(ulong source, ulong target) => $"({source}->{target})";
73
74          public static implicit operator ulong[](UInt64Link link) => link.ToArray();
75
76          public static implicit operator UInt64Link(ulong[] linkArray) => new
            ↪  UInt64Link(linkArray);
77
78          public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
            ↪  : ToString(Index, Source, Target);
79
80          #region IList
81
82          public ulong this[int index]
83          {
84              get
85              {
86                  Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
                    ↪  nameof(index));
87                  if (index == _constants.IndexPart)
88                  {
89                      return Index;
90                  }
91                  if (index == _constants.SourcePart)
92                  {
93                      return Source;
94                  }
95                  if (index == _constants.TargetPart)
96                  {
```

```
                    return Target;
                }
                throw new NotSupportedException(); // Impossible path due to Ensure.ArgumentInRange
            }
            set => throw new NotSupportedException();
        }

        public int Count => Length;

        public bool IsReadOnly => true;

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

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

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

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

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

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

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

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

            return -1;
        }

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

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

        #endregion
    }
}
```

./Platform.Data.Doublets/UInt64LinkExtensions.cs

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

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

```
}
```

./Platform.Data.Doublets/UInt64LinksExtensions.cs

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

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

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

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

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

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

        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<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
        {
            var sb = new StringBuilder();
            var visited = new HashSet<ulong>();
            links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
            ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
            return sb.ToString();
```

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

```
                    {
                        sb.Append('*');
                    }
                    sb.Append(linkIndex);
                }
            }
            else
            {
                if (renderDebug)
                {
                    sb.Append('~');
                }
                sb.Append(linkIndex);
            }
        }
    }
}
```

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

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

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

namespace Platform.Data.Doublets
{
    public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
    {
        /// <remarks>
        /// Альтернативные варианты хранения трансформации (элемента транзакции):
        ///
        /// private enum TransitionType
        /// {
        ///     Creation,
        ///     UpdateOf,
        ///     UpdateTo,
        ///     Deletion
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public ulong TransactionId;
        ///     public UniqueTimestamp Timestamp;
        ///     public TransactionItemType Type;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
        /// }
        ///
        /// Или
        ///
        /// public struct TransitionHeader
        /// {
        ///     public ulong TransactionIdCombined;
        ///     public ulong TimestampCombined;
        ///
        ///     public ulong TransactionId
        ///     {
        ///         get
        ///         {
        ///             return (ulong) mask &amp; TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public UniqueTimestamp Timestamp
        ///     {
        ///         get
        ///         {
```

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

    public readonly ulong TransactionId;
    public readonly UInt64Link Before;
    public readonly UInt64Link After;
    public readonly Timestamp Timestamp;

    public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId, UInt64Link before, UInt64Link after)
    {
        TransactionId = transactionId;
        Before = before;
        After = after;
        Timestamp = uniqueTimestampFactory.Create();
    }

    public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId, UInt64Link before)
        : this(uniqueTimestampFactory, transactionId, before, default)
    {
    }

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

    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; }
            public bool IsReverted { get; private set; }

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

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

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

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

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

            protected override void Dispose(bool manual, bool wasDisposed)
            {
```

```
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;
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 UInt64Link(Links.GetLink(createdLinkIndex));
273             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  default, createdLink));
274             return createdLinkIndex;
275         }
276 
277         public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
278         {
279             var linkIndex = restrictions[Constants.IndexPart];
280             var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
281             linkIndex = Links.Update(restrictions, substitution);
282             var afterLink = new UInt64Link(Links.GetLink(linkIndex));
283             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  beforeLink, afterLink));
284             return linkIndex;
```

```
        }

        public override void Delete(IList<ulong> restrictions)
        {
            var link = restrictions[Constants.IndexPart];
            var deletedLink = new UInt64Link(Links.GetLink(link));
            Links.Delete(link);
            CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
            ↪  deletedLink, default));
        }

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

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

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

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

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

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

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

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

        private void DisposeTransitions()
        {
            try
            {
```

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

        #region DisposalBase

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

        #endregion
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Unicode
{
    public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<char, TLink>
    {
        private readonly IConverter<TLink> _addressToNumberConverter;
        private readonly TLink _unicodeSymbolMarker;

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

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

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

```
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Interfaces;
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Unicode
{
    public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<string, TLink>
    {
        private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
        private readonly ISequenceIndex<TLink> _index;
        private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
        private readonly TLink _unicodeSequenceMarker;

```

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

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

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

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

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

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

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

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

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

        public void Init()
        {
            if (_initialized)
            {
                return;
            }
            _initialized = true;
            var firstLink = _links.CreatePoint();
            if (firstLink != FirstCharLink)
            {
                _links.Delete(firstLink);
            }
            else
            {
                for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
                {
                    // From NIL to It (NIL -> Character) transformation meaning, (or infinite
                    ↪  amount of NIL characters before actual Character)
                    var createdLink = _links.CreatePoint();
                    _links.Update(createdLink, firstLink, createdLink);
                    if (createdLink != i)
                    {
                        throw new InvalidOperationException("Unable to initialize UTF 16
                        ↪  table.");
                    }
                }
```

```
            }
        }

        // 0 - null link
        // 1 - nil character (0 character)
        // ...
        // 65536 (0(1) + 65535 = 65536 possible values)

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong FromCharToLink(char character) => (ulong)character + 1;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static char FromLinkToChar(ulong link) => (char)(link - 1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsCharLink(ulong link) => link <= MapSize;

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

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

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

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

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

        public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
        {
            var result = new List<ulong[]>();
            var offset = 0;
            while (offset < sequence.Length)
            {
                var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
                var relativeLength = 1;
                var absoluteLength = offset + relativeLength;
                while (absoluteLength < sequence.Length &&
```

```
                        currentCategory ==
                        ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
                {
                    relativeLength++;
                    absoluteLength++;
                }
                // char array to ulong array
                var innerSequence = new ulong[relativeLength];
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = FromCharToLink(sequence[i]);
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }

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

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

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

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

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

```
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      {
```

```
        private readonly IConverter<TLink> _numberToAddressConverter;
        private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;

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

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

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

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

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

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

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

            ulong x = 10;
            ulong y = 500;

            bool result = false;

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

            var comparer1 = Comparer<ulong>.Default;

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

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

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

            var comparer2 = new UInt64Comparer();
```

```

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

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

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

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

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

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

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

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

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

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

            ulong x = 10;
            ulong y = 500;

            bool result = false;

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

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

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

            var equalityComparer1 = EqualityComparer<ulong>.Default;

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

```
66
67              var equalityComparer2 = new UInt64EqualityComparer();
68
69              var ts5 = Performance.Measure(() =>
70              {
71                  for (int i = 0; i < N; i++)
72                  {
73                      result = equalityComparer2.Equals(x, y);
74                  }
75              });
76
77              Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;
78
79              var ts6 = Performance.Measure(() =>
80              {
81                  for (int i = 0; i < N; i++)
82                  {
83                      result = equalityComparer3(x, y);
84                  }
85              });
86
87              var comparer = Comparer<ulong>.Default;
88
89              var ts7 = Performance.Measure(() =>
90              {
91                  for (int i = 0; i < N; i++)
92                  {
93                      result = comparer.Compare(x, y) == 0;
94                  }
95              });
96
97              Assert.True(ts2 < ts1);
98              Assert.True(ts3 < ts2);
99              Assert.True(ts5 < ts4);
100             Assert.True(ts5 < ts6);
101
102             Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");
103         }
104     }
105 }
```

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

```
1  using System;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Memory;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.ResizableDirectMemory;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public unsafe static class GenericLinksTests
11     {
12         [Fact]
13         public static void CRUDTest()
14         {
15             Using<byte>(links => links.TestCRUDOperations());
16             Using<ushort>(links => links.TestCRUDOperations());
17             Using<uint>(links => links.TestCRUDOperations());
18             Using<ulong>(links => links.TestCRUDOperations());
19         }
20
21         [Fact]
22         public static void RawNumbersCRUDTest()
23         {
24             Using<byte>(links => links.TestRawNumbersCRUDOperations());
25             Using<ushort>(links => links.TestRawNumbersCRUDOperations());
26             Using<uint>(links => links.TestRawNumbersCRUDOperations());
27             Using<ulong>(links => links.TestRawNumbersCRUDOperations());
28         }
29
30         [Fact]
31         public static void MultipleRandomCreationsAndDeletionsTest()
32         {
33             Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
                ↪ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
                ↪ implementation of tree cuts out 5 bits from the address space.
34             Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
                ↪ stMultipleRandomCreationsAndDeletions(100));
```

```
            Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
            Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
        }

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

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

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

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

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

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);

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

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

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

```

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

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

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);

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

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

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

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

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

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

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

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

            var optimalVariant = optimalVariantConverter.Convert(sequence);

            var readSequence1 = sequences.ToList(optimalVariant);

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

./Platform.Data.Doublets.Tests/ReadSequenceTests.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences;

namespace Platform.Data.Doublets.Tests
{
    public static class ReadSequenceTests
    {
        [Fact]
        public static void ReadSequenceTest()
        {
            const long sequenceLength = 2000;

            using (var scope = new TempLinksTestScope(useSequences: false))
```

```
            {
                var links = scope.Links;
                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong> {
                ↪  Walker = new LeveledSequenceWalker<ulong>(links) });

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

                var sw1 = Stopwatch.StartNew();
                var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();

                var sw2 = Stopwatch.StartNew();
                var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();

                var sw3 = Stopwatch.StartNew();
                var readSequence2 = new List<ulong>();
                SequenceWalker.WalkRight(balancedVariant,
                                         links.GetSource,
                                         links.GetTarget,
                                         links.IsPartialPoint,
                                         readSequence2.Add);
                sw3.Stop();

                Assert.True(sequence.SequenceEqual(readSequence1));

                Assert.True(sequence.SequenceEqual(readSequence2));

                // Assert.True(sw2.Elapsed < sw3.Elapsed);

                Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
                ↪  {sw2.Elapsed}");

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }
    }
}
```

./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs

```
using System.IO;
using Xunit;
using Platform.Singletons;
using Platform.Memory;
using Platform.Data.Doublets.ResizableDirectMemory;

namespace Platform.Data.Doublets.Tests
{
    public static class ResizableDirectMemoryLinksTests
    {
        private static readonly LinksConstants<ulong> _constants =
        ↪  Default<LinksConstants<ulong>>.Instance;

        [Fact]
        public static void BasicFileMappedMemoryTest()
        {
            var tempFilename = Path.GetTempFileName();
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
            File.Delete(tempFilename);
        }

        [Fact]
        public static void BasicHeapMemoryTest()
        {
            using (var memory = new
            ↪  HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
            ↪  UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            {
```

```
                memoryAdapter.TestBasicMemoryOperations();
            }
        }

        private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Delete(link);
        }

        [Fact]
        public static void NonexistentReferencesHeapMemoryTest()
        {
            using (var memory = new
            ↪  HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
            ↪  UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestNonexistentReferences();
            }
        }

        private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
            var resultLink = _constants.Null;
            memoryAdapter.Each(foundLink =>
            {
                resultLink = foundLink[_constants.IndexPart];
                return _constants.Break;
            }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
            Assert.True(resultLink == link);
            Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
            memoryAdapter.Delete(link);
        }
    }
}
```

./Platform.Data.Doublets.Tests/ScopeTests.cs

```
using Xunit;
using Platform.Scopes;
using Platform.Memory;
using Platform.Data.Doublets.ResizableDirectMemory;
using Platform.Data.Doublets.Decorators;
using Platform.Reflection;

namespace Platform.Data.Doublets.Tests
{
    public static class ScopeTests
    {
        [Fact]
        public static void SingleDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.IncludeAssemblyOf<IMemory>();
                var instance = scope.Use<IDirectMemory>();
                Assert.IsType<HeapResizableDirectMemory>(instance);
            }
        }

        [Fact]
        public static void CascadeDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.Include<TemporaryFileMappedResizableDirectMemory>();
                scope.Include<UInt64ResizableDirectMemoryLinks>();
                var instance = scope.Use<ILinks<ulong>>();
                Assert.IsType<UInt64ResizableDirectMemoryLinks>(instance);
            }
        }

        [Fact]
        public static void FullAutoResolutionTest()
        {
            using (var scope = new Scope(autoInclude: true, autoExplore: true))
            {
```

```
                var instance = scope.Use<UInt64Links>();
                Assert.IsType<UInt64Links>(instance);
            }
        }

        [Fact]
        public static void TypeParametersTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  ResizableDirectMemoryLinks<ulong>>>())
            {
                var links = scope.Use<ILinks<ulong>>();
                Assert.IsType<ResizableDirectMemoryLinks<ulong>>(links);
            }
        }
    }
}
```

./Platform.Data.Doublets.Tests/SequencesTests.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Collections;
using Platform.Random;
using Platform.IO;
using Platform.Singletons;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Unicode;

namespace Platform.Data.Doublets.Tests
{
    public static class SequencesTests
    {
        private static readonly LinksConstants<ulong> _constants =
        ↪  Default<LinksConstants<ulong>>.Instance;

        static SequencesTests()
        {
            // Trigger static constructor to not mess with perfomance measurements
            _ = BitString.GetBitMaskFromIndex(1);
        }

        [Fact]
        public static void CreateAllVariantsTest()
        {
            const long sequenceLength = 8;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                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);
            }
        }
```

```
61
62          //[Fact]
63          //public void CUDTest()
64          //{
65          //    var tempFilename = Path.GetTempFileName();
66
67          //    const long sequenceLength = 8;
68
69          //    const ulong itself = LinksConstants.Itself;
70
71          //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
            ↪  DefaultLinksSizeStep))
72          //    using (var links = new Links(memoryAdapter))
73          //    {
74          //        var sequence = new ulong[sequenceLength];
75          //        for (var i = 0; i < sequenceLength; i++)
76          //            sequence[i] = links.Create(itself, itself);
77
78          //        SequencesOptions o = new SequencesOptions();
79
80          // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
81          //        o.
82
83          //        var sequences = new Sequences(links);
84
85          //        var sw1 = Stopwatch.StartNew();
86          //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
87
88          //        var sw2 = Stopwatch.StartNew();
89          //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
90
91          //        Assert.True(results1.Count > results2.Length);
92          //        Assert.True(sw1.Elapsed > sw2.Elapsed);
93
94          //        for (var i = 0; i < sequenceLength; i++)
95          //            links.Delete(sequence[i]);
96          //    }
97
98          //    File.Delete(tempFilename);
99          //}
100
101         [Fact]
102         public static void AllVariantsSearchTest()
103         {
104             const long sequenceLength = 8;
105
106             using (var scope = new TempLinksTestScope(useSequences: true))
107             {
108                 var links = scope.Links;
109                 var sequences = scope.Sequences;
110
111                 var sequence = new ulong[sequenceLength];
112                 for (var i = 0; i < sequenceLength; i++)
113                 {
114                     sequence[i] = links.Create();
115                 }
116
117                 var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
118
119                 //for (int i = 0; i < createResults.Length; i++)
120                 //    sequences.Create(createResults[i]);
121
122                 var sw0 = Stopwatch.StartNew();
123                 var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
124
125                 var sw1 = Stopwatch.StartNew();
126                 var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
127
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();
```

```
            Assert.True(intersection1.Count == searchResults1.Count);
            Assert.True(intersection1.Count == createResults.Length);

            var intersection2 = createResults.Intersect(searchResults2).ToList();
            Assert.True(intersection2.Count == searchResults2.Count);
            Assert.True(intersection2.Count == createResults.Length);

            var intersection3 = createResults.Intersect(searchResults3).ToList();
            Assert.True(intersection3.Count == searchResults3.Count);
            Assert.True(intersection3.Count == createResults.Length);

            for (var i = 0; i < sequenceLength; i++)
            {
                links.Delete(sequence[i]);
            }
        }
    }

    [Fact]
    public static void BalancedVariantSearchTest()
    {
        const long sequenceLength = 200;

        using (var scope = new TempLinksTestScope(useSequences: true))
        {
            var links = scope.Links;
            var sequences = scope.Sequences;

            var sequence = new ulong[sequenceLength];
            for (var i = 0; i < sequenceLength; i++)
            {
                sequence[i] = links.Create();
            }

            var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

            var sw1 = Stopwatch.StartNew();
            var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();

            var sw2 = Stopwatch.StartNew();
            var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();

            var sw3 = Stopwatch.StartNew();
            var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();

            // На количестве в 200 элементов это будет занимать вечность
            //var sw4 = Stopwatch.StartNew();
            //var searchResults4 = sequences.Each(sequence); sw4.Stop();

            Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);

            Assert.True(searchResults3.Count == 1 && balancedVariant ==
            ↪  searchResults3.First());

            //Assert.True(sw1.Elapsed < sw2.Elapsed);

            for (var i = 0; i < sequenceLength; i++)
            {
                links.Delete(sequence[i]);
            }
        }
    }

    [Fact]
    public static void AllPartialVariantsSearchTest()
    {
        const long sequenceLength = 8;

        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);
```

```
218
219                 //var createResultsStrings = createResults.Select(x => x + ": " +
                    ↪ sequences.FormatSequence(x)).ToList();
220                 //Global.Trash = createResultsStrings;
221
222                 var partialSequence = new ulong[sequenceLength - 2];
223
224                 Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
225
226                 var sw1 = Stopwatch.StartNew();
227                 var searchResults1 =
                    ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
228
229                 var sw2 = Stopwatch.StartNew();
230                 var searchResults2 =
                    ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
231
232                 //var sw3 = Stopwatch.StartNew();
233                 //var searchResults3 =
                    ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
234
235                 var sw4 = Stopwatch.StartNew();
236                 var searchResults4 =
                    ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
237
238                 //Global.Trash = searchResults3;
239
240                 //var searchResults1Strings = searchResults1.Select(x => x + ": " +
                    ↪ sequences.FormatSequence(x)).ToList();
241                 //Global.Trash = searchResults1Strings;
242
243                 var intersection1 = createResults.Intersect(searchResults1).ToList();
244                 Assert.True(intersection1.Count == createResults.Length);
245
246                 var intersection2 = createResults.Intersect(searchResults2).ToList();
247                 Assert.True(intersection2.Count == createResults.Length);
248
249                 var intersection4 = createResults.Intersect(searchResults4).ToList();
250                 Assert.True(intersection4.Count == createResults.Length);
251
252                 for (var i = 0; i < sequenceLength; i++)
253                 {
254                     links.Delete(sequence[i]);
255                 }
256             }
257         }
258
259         [Fact]
260         public static void BalancedPartialVariantsSearchTest()
261         {
262             const long sequenceLength = 200;
263
264             using (var scope = new TempLinksTestScope(useSequences: true))
265             {
266                 var links = scope.Links;
267                 var sequences = scope.Sequences;
268
269                 var sequence = new ulong[sequenceLength];
270                 for (var i = 0; i < sequenceLength; i++)
271                 {
272                     sequence[i] = links.Create();
273                 }
274
275                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
276
277                 var balancedVariant = balancedVariantConverter.Convert(sequence);
278
279                 var partialSequence = new ulong[sequenceLength - 2];
280
281                 Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
282
283                 var sw1 = Stopwatch.StartNew();
284                 var searchResults1 =
                    ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
285
286                 var sw2 = Stopwatch.StartNew();
287                 var searchResults2 =
                    ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
288
```

```
289                 Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
290
291                 Assert.True(searchResults2.Count == 1 && balancedVariant ==
                    ↪ searchResults2.First());
292
293                 for (var i = 0; i < sequenceLength; i++)
294                 {
295                     links.Delete(sequence[i]);
296                 }
297             }
298         }
299
300         [Fact(Skip = "Correct implementation is pending")]
301         public static void PatternMatchTest()
302         {
303             var zeroOrMany = Sequences.Sequences.ZeroOrMany;
304
305             using (var scope = new TempLinksTestScope(useSequences: true))
306             {
307                 var links = scope.Links;
308                 var sequences = scope.Sequences;
309
310                 var e1 = links.Create();
311                 var e2 = links.Create();
312
313                 var sequence = new[]
314                 {
315                     e1, e2, e1, e2 // mama / papa
316                 };
317
318                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
319
320                 var balancedVariant = balancedVariantConverter.Convert(sequence);
321
322                 // 1: [1]
323                 // 2: [2]
324                 // 3: [1,2]
325                 // 4: [1,2,1,2]
326
327                 var doublet = links.GetSource(balancedVariant);
328
329                 var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
330
331                 Assert.True(matchedSequences1.Count == 0);
332
333                 var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
334
335                 Assert.True(matchedSequences2.Count == 0);
336
337                 var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
338
339                 Assert.True(matchedSequences3.Count == 0);
340
341                 var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
342
343                 Assert.Contains(doublet, matchedSequences4);
344                 Assert.Contains(balancedVariant, matchedSequences4);
345
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                 {
```

```
                    e1, e2, e1, e2 // mama / papa
                };

                Assert.False(index.MightContain(sequence));

                index.Add(sequence);

                Assert.True(index.MightContain(sequence));
            }
        }

        /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87%D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
        private static readonly string _exampleText =
            @"([english version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там где есть место для нового начала? Разве пустота это не характеристика пространства? Пространство это то, что можно чем-то наполнить?

[![чёрное пространство, белое пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png ""чёрное пространство, белое пространство"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png)

Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?

[![чёрное пространство, чёрная точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png ""чёрное пространство, чёрная точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)

А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть так? Инверсия? Отражение? Сумма?

[![белая точка, чёрная точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png ""белая точка, чёрная точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)

А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой? Гранью? Разделителем? Единицей?

[![две белые точки, чёрная вертикальная линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png ""две белые точки, чёрная вертикальная линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)

Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец? Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если у него нет размера? Будет ли круг точкой? Точка состоящая из точек?

[![белая вертикальная линия, чёрный круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png ""белая вертикальная линия, чёрный круг"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)

Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально? Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли элементарная единица смысла?

[![белый круг, чёрная горизонтальная линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png ""белый круг, чёрная горизонтальная линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)

Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла ""соединить, связать"", есть ещё и смысл направления ""от начала к концу""? От предка к потомку? От родителя к ребёнку? От общего к частному?
```

```
[![белая горизонтальная линия, чёрная горизонтальная стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png ""белая горизонтальная линия, чёрная горизонтальная стрелка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)

Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два объекта, как бы это выглядело?

[![белая связь, чёрная направленная связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png ""белая связь, чёрная направленная связь"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)

Допустим у нас есть смысл ""связать"" и смысл ""направления"", много ли это нам даёт? Много ли вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие? Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в его конечном состоянии, если конечно конец определён направлением?

[![белая обычная и направленная связи, чёрная типизированная связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png ""белая обычная и направленная связи, чёрная типизированная связь"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)

А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри? Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?

[![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная связь с рекурсивной внутренней структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png ""белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная связь с рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)

На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом рекурсии или фрактала?

[![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная типизированная связь с двойной рекурсивной внутренней структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png ""белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная типизированная связь с двойной рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)

Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы? Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?

[![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png ""белая обычная и направленная связи со структурой из 8 цветных элементов последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последовательности"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)

...

[![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif)";

        private static readonly string _exampleLoremIpsumText =
            @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.";

        [Fact]
        public static void CompressionTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var e1 = links.Create();
```

```
447                 var e2 = links.Create();
448
449                 var sequence = new[]
450                 {
451                     e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
452                 };
453
454                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
455                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
456                 var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
457                 var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
                    ↪  balancedVariantConverter, doubletFrequenciesCache);
458
459                 var compressedVariant = compressingConverter.Convert(sequence);
460
461                 // 1: [1]       (1->1) point
462                 // 2: [2]       (2->2) point
463                 // 3: [1,2]     (1->2) doublet
464                 // 4: [1,2,1,2] (3->3) doublet
465
466                 Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
467                 Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
468                 Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
469                 Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
470
471                 var source = _constants.SourcePart;
472                 var target = _constants.TargetPart;
473
474                 Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
475                 Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
476                 Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
477                 Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
478
479                 // 4 - length of sequence
480                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
                    ↪  == sequence[0]);
481                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
                    ↪  == sequence[1]);
482                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
                    ↪  == sequence[2]);
483                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
                    ↪  == sequence[3]);
484             }
485         }
486
487         [Fact]
488         public static void CompressionEfficiencyTest()
489         {
490             var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
                ↪  StringSplitOptions.RemoveEmptyEntries);
491             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
492             var totalCharacters = arrays.Select(x => x.Length).Sum();
493
494             using (var scope1 = new TempLinksTestScope(useSequences: true))
495             using (var scope2 = new TempLinksTestScope(useSequences: true))
496             using (var scope3 = new TempLinksTestScope(useSequences: true))
497             {
498                 scope1.Links.Unsync.UseUnicode();
499                 scope2.Links.Unsync.UseUnicode();
500                 scope3.Links.Unsync.UseUnicode();
501
502                 var balancedVariantConverter1 = new
                    ↪  BalancedVariantConverter<ulong>(scope1.Links.Unsync);
503                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
504                 var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
505                 var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
                    ↪  balancedVariantConverter1, linkFrequenciesCache1,
                    ↪  doInitialFrequenciesIncrement: false);
506
507                 //var compressor2 = scope2.Sequences;
508                 var compressor3 = scope3.Sequences;
509
510                 var constants = Default<LinksConstants<ulong>>.Instance;
511
```

```
var sequences = compressor3;
//var meaningRoot = links.CreatePoint();
//var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
//var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
//var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
↪  constants.Itself);

//var unaryNumberToAddressConverter = new
↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
//var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
↪  unaryOne);
//var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
//var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
↪  frequencyPropertyMarker, frequencyMarker);
//var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
↪  frequencyPropertyOperator, frequencyIncrementer);
//var linkToItsFrequencyNumberConverter = new
↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
↪  unaryNumberToAddressConverter);

var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
↪  totalSequenceSymbolFrequencyCounter);

var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
↪  ncyNumberConverter<ulong>(linkFrequenciesCache3);

var sequenceToItsLocalElementLevelsConverter = new
↪  SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
↪  linkToItsFrequencyNumberConverter);
var optimalVariantConverter = new
↪  OptimalVariantConverter<ulong>(scope3.Links.Unsync,
↪  sequenceToItsLocalElementLevelsConverter);

var compressed1 = new ulong[arrays.Length];
var compressed2 = new ulong[arrays.Length];
var compressed3 = new ulong[arrays.Length];

var START = 0;
var END = arrays.Length;

//for (int i = START; i < END; i++)
//    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);

var initialCount1 = scope2.Links.Unsync.Count();

var sw1 = Stopwatch.StartNew();

for (int i = START; i < END; i++)
{
    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
    compressed1[i] = compressor1.Convert(arrays[i]);
}

var elapsed1 = sw1.Elapsed;

var balancedVariantConverter2 = new
↪  BalancedVariantConverter<ulong>(scope2.Links.Unsync);

var initialCount2 = scope2.Links.Unsync.Count();

var sw2 = Stopwatch.StartNew();

for (int i = START; i < END; i++)
{
    compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
}

var elapsed2 = sw2.Elapsed;

for (int i = START; i < END; i++)
{
    linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
}

var initialCount3 = scope3.Links.Unsync.Count();

var sw3 = Stopwatch.StartNew();

```

```
576                 for (int i = START; i < END; i++)
577                 {
578                     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
579                     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
580                 }
581
582                 var elapsed3 = sw3.Elapsed;
583
584                 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
                    ↪ Optimal variant: {elapsed3}");
585
586                 // Assert.True(elapsed1 > elapsed2);
587
588                 // Checks
589                 for (int i = START; i < END; i++)
590                 {
591                     var sequence1 = compressed1[i];
592                     var sequence2 = compressed2[i];
593                     var sequence3 = compressed3[i];
594
595                     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪ scope1.Links.Unsync);
596
597                     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪ scope2.Links.Unsync);
598
599                     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
                        ↪ scope3.Links.Unsync);
600
601                     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
                        ↪ link.IsPartialPoint());
602                     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
                        ↪ link.IsPartialPoint());
603                     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
                        ↪ link.IsPartialPoint());
604
605                     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                        ↪ arrays[i].Length > 3)
606                     //    Assert.False(structure1 == structure2);
607                     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
                        ↪ arrays[i].Length > 3)
608                     //    Assert.False(structure3 == structure2);
609
610                     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
611                     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
612                 }
613
614                 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
                    ↪ totalCharacters);
615                 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
                    ↪ totalCharacters);
616                 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
                    ↪ totalCharacters);
617
618                 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
                    ↪ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
                    ↪ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
                    ↪ totalCharacters}");
619
620                 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
                    ↪ scope2.Links.Unsync.Count() - initialCount2);
621                 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
                    ↪ scope2.Links.Unsync.Count() - initialCount2);
622
623                 var duplicateProvider1 = new
                    ↪ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
624                 var duplicateProvider2 = new
                    ↪ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
625                 var duplicateProvider3 = new
                    ↪ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
626
627                 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
628                 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
629                 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
630
631                 var duplicates1 = duplicateCounter1.Count();
632
```

```
633                 ConsoleHelpers.Debug("------");
634
635                 var duplicates2 = duplicateCounter2.Count();
636
637                 ConsoleHelpers.Debug("------");
638
639                 var duplicates3 = duplicateCounter3.Count();
640
641                 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
642
643                 linkFrequenciesCache1.ValidateFrequencies();
644                 linkFrequenciesCache3.ValidateFrequencies();
645             }
646         }
647
648         [Fact]
649         public static void CompressionStabilityTest()
650         {
651             // TODO: Fix bug (do a separate test)
652             //const ulong minNumbers = 0;
653             //const ulong maxNumbers = 1000;
654
655             const ulong minNumbers = 10000;
656             const ulong maxNumbers = 12500;
657
658             var strings = new List<string>();
659
660             for (ulong i = minNumbers; i < maxNumbers; i++)
661             {
662                 strings.Add(i.ToString());
663             }
664
665             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
666             var totalCharacters = arrays.Select(x => x.Length).Sum();
667
668             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                ↪  SequencesOptions<ulong> { UseCompression = true,
                ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
669             using (var scope2 = new TempLinksTestScope(useSequences: true))
670             {
671                 scope1.Links.UseUnicode();
672                 scope2.Links.UseUnicode();
673
674                 //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
675                 var compressor1 = scope1.Sequences;
676                 var compressor2 = scope2.Sequences;
677
678                 var compressed1 = new ulong[arrays.Length];
679                 var compressed2 = new ulong[arrays.Length];
680
681                 var sw1 = Stopwatch.StartNew();
682
683                 var START = 0;
684                 var END = arrays.Length;
685
686                 // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
687                 // Stability issue starts at 10001 or 11000
688                 //for (int i = START; i < END; i++)
689                 //{
690                 //    var first = compressor1.Compress(arrays[i]);
691                 //    var second = compressor1.Compress(arrays[i]);
692
693                 //    if (first == second)
694                 //        compressed1[i] = first;
695                 //    else
696                 //    {
697                 //        // TODO: Find a solution for this case
698                 //    }
699                 //}
700
701                 for (int i = START; i < END; i++)
702                 {
703                     var first = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
704                     var second = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
705
706                     if (first == second)
707                     {
708                         compressed1[i] = first;
709                     }
710                     else
```

```
                    {
                        // TODO: Find a solution for this case
                    }
                }

                var elapsed1 = sw1.Elapsed;

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);

                var sw2 = Stopwatch.StartNew();

                for (int i = START; i < END; i++)
                {
                    var first = balancedVariantConverter.Convert(arrays[i]);
                    var second = balancedVariantConverter.Convert(arrays[i]);

                    if (first == second)
                    {
                        compressed2[i] = first;
                    }
                }

                var elapsed2 = sw2.Elapsed;

                Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                ↪ {elapsed2}");

                Assert.True(elapsed1 > elapsed2);

                // Checks
                for (int i = START; i < END; i++)
                {
                    var sequence1 = compressed1[i];
                    var sequence2 = compressed2[i];

                    if (sequence1 != _constants.Null && sequence2 != _constants.Null)
                    {
                        var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪ scope1.Links);

                        var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪ scope2.Links);

                        //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                        ↪ link.IsPartialPoint());
                        //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                        ↪ link.IsPartialPoint());

                        //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                        ↪ arrays[i].Length > 3)
                        //    Assert.False(structure1 == structure2);

                        Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
                    }
                }

                Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
                Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);

                Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                ↪ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                ↪ totalCharacters}");

                Assert.True(scope1.Links.Count() <= scope2.Links.Count());

                //compressor1.ValidateFrequencies();
            }
        }

        [Fact]
        public static void RundomNumbersCompressionQualityTest()
        {
            const ulong N = 500;

            //const ulong minNumbers = 10000;
            //const ulong maxNumbers = 20000;

            //var strings = new List<string>();
```

```
782             //for (ulong i = 0; i < N; i++)
783             //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
                ↪  maxNumbers).ToString());
784
785             var strings = new List<string>();
786
787             for (ulong i = 0; i < N; i++)
788             {
789                 strings.Add(RandomHelpers.Default.NextUInt64().ToString());
790             }
791
792             strings = strings.Distinct().ToList();
793
794             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
795             var totalCharacters = arrays.Select(x => x.Length).Sum();
796
797             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                ↪  SequencesOptions<ulong> { UseCompression = true,
                ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
798             using (var scope2 = new TempLinksTestScope(useSequences: true))
799             {
800                 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
846                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪  scope2.Links);
847
848                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
849                     }
850                 }
851
852                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
853                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
854
```

```
855                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters}");
856
857                 // Can be worse than balanced variant
858                 //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
859
860                 //compressor1.ValidateFrequencies();
861             }
862         }
863
864         [Fact]
865         public static void AllTreeBreakDownAtSequencesCreationBugTest()
866         {
867             // Made out of AllPossibleConnectionsTest test.
868
869             //const long sequenceLength = 5; //100% bug
870             const long sequenceLength = 4; //100% bug
871             //const long sequenceLength = 3; //100% _no_bug (ok)
872
873             using (var scope = new TempLinksTestScope(useSequences: true))
874             {
875                 var links = scope.Links;
876                 var sequences = scope.Sequences;
877
878                 var sequence = new ulong[sequenceLength];
879                 for (var i = 0; i < sequenceLength; i++)
880                 {
881                     sequence[i] = links.Create();
882                 }
883
884                 var createResults = sequences.CreateAllVariants2(sequence);
885
886                 Global.Trash = createResults;
887
888                 for (var i = 0; i < sequenceLength; i++)
889                 {
890                     links.Delete(sequence[i]);
891                 }
892             }
893         }
894
895         [Fact]
896         public static void AllPossibleConnectionsTest()
897         {
898             const long sequenceLength = 5;
899
900             using (var scope = new TempLinksTestScope(useSequences: true))
901             {
902                 var links = scope.Links;
903                 var sequences = scope.Sequences;
904
905                 var sequence = new ulong[sequenceLength];
906                 for (var i = 0; i < sequenceLength; i++)
907                 {
908                     sequence[i] = links.Create();
909                 }
910
911                 var createResults = sequences.CreateAllVariants2(sequence);
912                 var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
913
914                 for (var i = 0; i < 1; i++)
915                 {
916                     var sw1 = Stopwatch.StartNew();
917                     var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
918
919                     var sw2 = Stopwatch.StartNew();
920                     var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
921
922                     var sw3 = Stopwatch.StartNew();
923                     var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
924
925                     var sw4 = Stopwatch.StartNew();
926                     var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
927
928                     Global.Trash = searchResults3;
929                     Global.Trash = searchResults4; //-V3008
930
931                     var intersection1 = createResults.Intersect(searchResults1).ToList();
932                     Assert.True(intersection1.Count == createResults.Length);
```

```
933
934                     var intersection2 = reverseResults.Intersect(searchResults1).ToList();
935                     Assert.True(intersection2.Count == reverseResults.Length);
936
937                     var intersection0 = searchResults1.Intersect(searchResults2).ToList();
938                     Assert.True(intersection0.Count == searchResults2.Count);
939
940                     var intersection3 = searchResults2.Intersect(searchResults3).ToList();
941                     Assert.True(intersection3.Count == searchResults3.Count);
942
943                     var intersection4 = searchResults3.Intersect(searchResults4).ToList();
944                     Assert.True(intersection4.Count == searchResults4.Count);
945                 }
946
947                 for (var i = 0; i < sequenceLength; i++)
948                 {
949                     links.Delete(sequence[i]);
950                 }
951             }
952         }
953
954         [Fact(Skip = "Correct implementation is pending")]
955         public static void CalculateAllUsagesTest()
956         {
957             const long sequenceLength = 3;
958
959             using (var scope = new TempLinksTestScope(useSequences: true))
960             {
961                 var links = scope.Links;
962                 var sequences = scope.Sequences;
963
964                 var sequence = new ulong[sequenceLength];
965                 for (var i = 0; i < sequenceLength; i++)
966                 {
967                     sequence[i] = links.Create();
968                 }
969
970                 var createResults = sequences.CreateAllVariants2(sequence);
971
972                 //var reverseResults =
                    ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
973
974                 for (var i = 0; i < 1; i++)
975                 {
976                     var linksTotalUsages1 = new ulong[links.Count() + 1];
977
978                     sequences.CalculateAllUsages(linksTotalUsages1);
979
980                     var linksTotalUsages2 = new ulong[links.Count() + 1];
981
982                     sequences.CalculateAllUsages2(linksTotalUsages2);
983
984                     var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
985                     Assert.True(intersection1.Count == linksTotalUsages2.Length);
986                 }
987
988                 for (var i = 0; i < sequenceLength; i++)
989                 {
990                     links.Delete(sequence[i]);
991                 }
992             }
993         }
994     }
995 }
```

./Platform.Data.Doublets.Tests/TempLinksTestScope.cs

```
1  using System.IO;
2  using Platform.Disposables;
3  using Platform.Data.Doublets.ResizableDirectMemory;
4  using Platform.Data.Doublets.Sequences;
5  using Platform.Data.Doublets.Decorators;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public class TempLinksTestScope : DisposableBase
10     {
11         public ILinks<ulong> MemoryAdapter { get; }
12         public SynchronizedLinks<ulong> Links { get; }
13         public Sequences.Sequences Sequences { get; }
14         public string TempFilename { get; }
```

```
15          public string TempTransactionLogFilename { get; }
16          private readonly bool _deleteFiles;
17
18          public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
            ↪  useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
            ↪  useLog) { }
19
20          public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
            ↪  true, bool useSequences = false, bool useLog = false)
21          {
22              _deleteFiles = deleteFiles;
23              TempFilename = Path.GetTempFileName();
24              TempTransactionLogFilename = Path.GetTempFileName();
25              var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
26              MemoryAdapter = useLog ? (ILinks<ulong>)new
                ↪  UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
                ↪  coreMemoryAdapter;
27              Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
28              if (useSequences)
29              {
30                  Sequences = new Sequences.Sequences(Links, sequencesOptions);
31              }
32          }
33
34          protected override void Dispose(bool manual, bool wasDisposed)
35          {
36              if (!wasDisposed)
37              {
38                  Links.Unsync.DisposeIfPossible();
39                  if (_deleteFiles)
40                  {
41                      DeleteFiles();
42                  }
43              }
44          }
45
46          public void DeleteFiles()
47          {
48              File.Delete(TempFilename);
49              File.Delete(TempTransactionLogFilename);
50          }
51      }
52  }
```

./Platform.Data.Doublets.Tests/TestExtensions.cs

```
1   using System.Collections.Generic;
2   using Xunit;
3   using Platform.Ranges;
4   using Platform.Numbers;
5   using Platform.Random;
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));
```

```
            // Get first link
            setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter.Result, linkAddress));

            // Update link to reference itself
            links.Update(linkAddress, linkAddress, linkAddress);

            link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(equalityComparer.Equals(link.Source, linkAddress));
            Assert.True(equalityComparer.Equals(link.Target, linkAddress));

            // Update link to reference null (prepare for delete)
            var updated = links.Update(linkAddress, constants.Null, constants.Null);

            Assert.True(equalityComparer.Equals(updated, linkAddress));

            link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(equalityComparer.Equals(link.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link.Target, constants.Null));

            // Delete link
            links.Delete(linkAddress);

            Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));

            setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
        }

        public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
        {
            // Constants
            var constants = links.Constants;
            var equalityComparer = EqualityComparer<T>.Default;

            var h106E = new Hybrid<T>(106L, isExternal: true);
            var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
            var h108E = new Hybrid<T>(-108L);

            Assert.Equal(106L, h106E.AbsoluteValue);
            Assert.Equal(107L, h107E.AbsoluteValue);
            Assert.Equal(108L, h108E.AbsoluteValue);

            // Create Link (External -> External)
            var linkAddress1 = links.Create();

            links.Update(linkAddress1, h106E, h108E);

            var link1 = new Link<T>(links.GetLink(linkAddress1));

            Assert.True(equalityComparer.Equals(link1.Source, h106E));
            Assert.True(equalityComparer.Equals(link1.Target, h108E));

            // Create Link (Internal -> External)
            var linkAddress2 = links.Create();

            links.Update(linkAddress2, linkAddress1, h108E);

            var link2 = new Link<T>(links.GetLink(linkAddress2));

            Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
            Assert.True(equalityComparer.Equals(link2.Target, h108E));

            // Create Link (Internal -> Internal)
            var linkAddress3 = links.Create();

            links.Update(linkAddress3, linkAddress1, linkAddress2);

            var link3 = new Link<T>(links.GetLink(linkAddress3));

            Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
            Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));

```

```
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

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Xunit;
using Platform.Disposables;
using Platform.IO;
using Platform.Ranges;
using Platform.Random;
using Platform.Timestamps;
using Platform.Reflection;
using Platform.Singletons;
using Platform.Scopes;
using Platform.Counters;
using Platform.Diagnostics;
using Platform.Memory;
using Platform.Data.Doublets.ResizableDirectMemory;
using Platform.Data.Doublets.Decorators;

namespace Platform.Data.Doublets.Tests
{
    public static class UInt64LinksTests
    {
        private static readonly LinksConstants<ulong> _constants = Default<LinksConstants<ulong>>.Instance;

        private const long Iterations = 10 * 1024;

        #region Concept

        [Fact]
        public static void MultipleCreateAndDeleteTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory, UInt64ResizableDirectMemoryLinks>>())
            {
                new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeletions(100);
            }
        }

        [Fact]
        public static void CascadeUpdateTest()
        {
            var itself = _constants.Itself;

            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;

                var l1 = links.Create();
                var l2 = links.Create();

                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>(scope.TempTransactionLogFilename);
            }
        }

        [Fact]
        public static void BasicTransactionLogTest()
        {
            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;
```

```
77                  var l1 = links.Create();
78                  var l2 = links.Create();
79
80                  Global.Trash = links.Update(l2, l2, l1, l2);
81
82                  links.Delete(l1);
83
84                  links.Unsync.DisposeIfPossible(); // Close links to access log
85
86                  Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    ↪  e.TempTransactionLogFilename);
87              }
88          }
89
90          [Fact]
91          public static void TransactionAutoRevertedTest()
92          {
93              // Auto Reverted (Because no commit at transaction)
94              using (var scope = new TempLinksTestScope(useLog: true))
95              {
96                  var links = scope.Links;
97                  var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
98                  using (var transaction = transactionsLayer.BeginTransaction())
99                  {
100                     var l1 = links.Create();
101                     var l2 = links.Create();
102
103                     links.Update(l2, l2, l1, l2);
104                 }
105
106                 Assert.Equal(0UL, links.Count());
107
108                 links.Unsync.DisposeIfPossible();
109
110                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
                    ↪  cope.TempTransactionLogFilename);
111                 Assert.Single(transitions);
112             }
113         }
114
115         [Fact]
116         public static void TransactionUserCodeErrorNoDataSavedTest()
117         {
118             // User Code Error (Autoreverted), no data saved
119             var itself = _constants.Itself;
120
121             TempLinksTestScope lastScope = null;
122             try
123             {
124                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪  useLog: true))
125                 {
126                     var links = scope.Links;
127                     var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
                        ↪  atorBase<ulong>)links.Unsync).Links;
128                     using (var transaction = transactionsLayer.BeginTransaction())
129                     {
130                         var l1 = links.CreateAndUpdate(itself, itself);
131                         var l2 = links.CreateAndUpdate(itself, itself);
132
133                         l2 = links.Update(l2, l2, l1, l2);
134
135                         links.CreateAndUpdate(l2, itself);
136                         links.CreateAndUpdate(l2, itself);
137
138                         //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
                            ↪  tion>(scope.TempTransactionLogFilename);
139
140                         l2 = links.Update(l2, l1);
141
142                         links.Delete(l2);
143
144                         ExceptionThrower();
145
146                         transaction.Commit();
147                     }
148
149                     Global.Trash = links.Count();
150                 }
```

```
151             }
152             catch
153             {
154                 Assert.False(lastScope == null);
155
156                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
                    ↪ astScope.TempTransactionLogFilename);
157
158                 Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
                    ↪ transitions[0].After.IsNull());
159
160                 lastScope.DeleteFiles();
161             }
162         }
163
164         [Fact]
165         public static void TransactionUserCodeErrorSomeDataSavedTest()
166         {
167             // User Code Error (Autoreverted), some data saved
168             var itself = _constants.Itself;
169
170             TempLinksTestScope lastScope = null;
171             try
172             {
173                 ulong l1;
174                 ulong l2;
175
176                 using (var scope = new TempLinksTestScope(useLog: true))
177                 {
178                     var links = scope.Links;
179                     l1 = links.CreateAndUpdate(itself, itself);
180                     l2 = links.CreateAndUpdate(itself, itself);
181
182                     l2 = links.Update(l2, l2, l1, l2);
183
184                     links.CreateAndUpdate(l2, itself);
185                     links.CreateAndUpdate(l2, itself);
186
187                     links.Unsync.DisposeIfPossible();
188
189                     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
                        ↪ scope.TempTransactionLogFilename);
190                 }
191
192                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪ useLog: true))
193                 {
194                     var links = scope.Links;
195                     var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
196                     using (var transaction = transactionsLayer.BeginTransaction())
197                     {
198                         l2 = links.Update(l2, l1);
199
200                         links.Delete(l2);
201
202                         ExceptionThrower();
203
204                         transaction.Commit();
205                     }
206
207                     Global.Trash = links.Count();
208                 }
209             }
210             catch
211             {
212                 Assert.False(lastScope == null);
213
214                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
                    ↪ Scope.TempTransactionLogFilename);
215
216                 lastScope.DeleteFiles();
217             }
218         }
219
220         [Fact]
221         public static void TransactionCommit()
222         {
223             var itself = _constants.Itself;
224
225             var tempDatabaseFilename = Path.GetTempFileName();
```

```
226             var tempTransactionLogFilename = Path.GetTempFileName();
227
228             // Commit
229             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                ↪  tempTransactionLogFilename))
230             using (var links = new UInt64Links(memoryAdapter))
231             {
232                 using (var transaction = memoryAdapter.BeginTransaction())
233                 {
234                     var l1 = links.CreateAndUpdate(itself, itself);
235                     var l2 = links.CreateAndUpdate(itself, itself);
236
237                     Global.Trash = links.Update(l2, l2, l1, l2);
238
239                     links.Delete(l1);
240
241                     transaction.Commit();
242                 }
243
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
314             {
315                 ulong l1;
316                 ulong l2;
317
318                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                    ↪ tempTransactionLogFilename))
319                 using (var links = new UInt64Links(memoryAdapter))
320                 {
321                     l1 = links.CreateAndUpdate(itself, itself);
322                     l2 = links.CreateAndUpdate(itself, itself);
323
324                     l2 = links.Update(l2, l2, l1, l2);
325
326                     links.CreateAndUpdate(l2, itself);
327                     links.CreateAndUpdate(l2, itself);
328                 }
329
330                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                    ↪ TransactionLogFilename);
331
332                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                    ↪ tempTransactionLogFilename))
333                 using (var links = new UInt64Links(memoryAdapter))
334                 {
335                     using (var transaction = memoryAdapter.BeginTransaction())
336                     {
337                         l2 = links.Update(l2, l1);
338
339                         links.Delete(l2);
340
341                         ExceptionThrower();
342
343                         transaction.Commit();
344                     }
345
346                     Global.Trash = links.Count();
347                 }
348             }
349             catch
350             {
351                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                    ↪ TransactionLogFilename);
352             }
353
354             File.Delete(tempDatabaseFilename);
355             File.Delete(tempTransactionLogFilename);
356         }
357
358         private static void ExceptionThrower() => throw new InvalidOperationException();
359
360         [Fact]
361         public static void PathsTest()
362         {
363             var source = _constants.SourcePart;
364             var target = _constants.TargetPart;
365
```

```
366             using (var scope = new TempLinksTestScope())
367             {
368                 var links = scope.Links;
369                 var l1 = links.CreatePoint();
370                 var l2 = links.CreatePoint();
371 
372                 var r1 = links.GetByKeys(l1, source, target, source);
373                 var r2 = links.CheckPathExistance(l2, l2, l2, l2);
374             }
375         }
376 
377         [Fact]
378         public static void RecursiveStringFormattingTest()
379         {
380             using (var scope = new TempLinksTestScope(useSequences: true))
381             {
382                 var links = scope.Links;
383                 var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
384 
385                 var a = links.CreatePoint();
386                 var b = links.CreatePoint();
387                 var c = links.CreatePoint();
388 
389                 var ab = links.CreateAndUpdate(a, b);
390                 var cb = links.CreateAndUpdate(c, b);
391                 var ac = links.CreateAndUpdate(a, c);
392 
393                 a = links.Update(a, c, b);
394                 b = links.Update(b, a, c);
395                 c = links.Update(c, a, b);
396 
397                 Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
398                 Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
399                 Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
400 
401                 Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                    ↪  "(5:(4:5 (6:5 4)) 6)");
402                 Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                    ↪  "(6:(5:(4:5 6) 6) 4)");
403                 Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
                    ↪  "(4:(5:4 (6:5 4)) 6)");
404 
405                 // TODO: Think how to build balanced syntax tree while formatting structure (eg.
                    ↪  "(4:(5:4 6) (6:5 4)") instead of "(4:(5:4 (6:5 4)) 6)"
406 
407                 Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
                    ↪  "{{5}{5}{4}{6}}");
408                 Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                    ↪  "{{5}{6}{6}{4}}");
409                 Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                    ↪  "{{4}{5}{4}{6}}");
410             }
411         }
412 
413         private static void DefaultFormatter(StringBuilder sb, ulong link)
414         {
415             sb.Append(link.ToString());
416         }
417 
418         #endregion
419 
420         #region Performance
421 
422         /*
423         public static void RunAllPerformanceTests()
424         {
425             try
426             {
427                 links.TestLinksInSteps();
428             }
429             catch (Exception ex)
430             {
431                 ex.WriteToConsole();
432             }
433 
434             return;
435 
436             try
437             {
```

```
438                 //ThreadPool.SetMaxThreads(2, 2);
439
440                 // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    ↪  результат
441                 // Также это дополнительно помогает в отладке
442                 // Увеличивает вероятность попадания информации в кэши
443                 for (var i = 0; i < 10; i++)
444                 {
445                     //0 - 10 ГБ
446                     //Каждые 100 МБ срез цифр
447
448                     //links.TestGetSourceFunction();
449                     //links.TestGetSourceFunctionInParallel();
450                     //links.TestGetTargetFunction();
451                     //links.TestGetTargetFunctionInParallel();
452                     links.Create64BillionLinks();
453
454                     links.TestRandomSearchFixed();
455                     //links.Create64BillionLinksInParallel();
456                     links.TestEachFunction();
457                     //links.TestForeach();
458                     //links.TestParallelForeach();
459                 }
460
461                 links.TestDeletionOfAllLinks();
462
463             }
464             catch (Exception ex)
465             {
466                 ex.WriteToConsole();
467             }
468         }*/
469
470          /*
471         public static void TestLinksInSteps()
472         {
473             const long gibibyte = 1024 * 1024 * 1024;
474             const long mebibyte = 1024 * 1024;
475
476             var totalLinksToCreate = gibibyte /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
477             var linksStep = 102 * mebibyte /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478
479             var creationMeasurements = new List<TimeSpan>();
480             var searchMeasuremets = new List<TimeSpan>();
481             var deletionMeasurements = new List<TimeSpan>();
482
483             GetBaseRandomLoopOverhead(linksStep);
484             GetBaseRandomLoopOverhead(linksStep);
485
486             var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
487
488             ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
489
490             var loops = totalLinksToCreate / linksStep;
491
492             for (int i = 0; i < loops; i++)
493             {
494                 creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
495                 searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));
496
497                 Console.Write("\rC + S {0}/{1}", i + 1, loops);
498             }
499
500             ConsoleHelpers.Debug();
501
502             for (int i = 0; i < loops; i++)
503             {
504                 deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
505
506                 Console.Write("\rD {0}/{1}", i + 1, loops);
507             }
508
509             ConsoleHelpers.Debug();
510
511             ConsoleHelpers.Debug("C S D");
512
513             for (int i = 0; i < loops; i++)
514             {
```

```
515                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
    ↪  searchMeasuremets[i], deletionMeasurements[i]);
516             }
517
518             ConsoleHelpers.Debug("C S D (no overhead)");
519
520             for (int i = 0; i < loops; i++)
521             {
522                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
    ↪  searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
523             }
524
525             ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
    ↪  links.Total);
526         }
527
528         private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
    ↪  amountToCreate)
529         {
530             for (long i = 0; i < amountToCreate; i++)
531                 links.Create(0, 0);
532         }
533
534          private static TimeSpan GetBaseRandomLoopOverhead(long loops)
535          {
536              return Measure(() =>
537              {
538                  ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
539                  ulong result = 0;
540                  for (long i = 0; i < loops; i++)
541                  {
542                      var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
543                      var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544
545                      result += maxValue + source + target;
546                  }
547                  Global.Trash = result;
548              });
549          }
550           */
551
552          [Fact(Skip = "performance test")]
553          public static void GetSourceTest()
554          {
555              using (var scope = new TempLinksTestScope())
556              {
557                  var links = scope.Links;
558                  ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
                     ↪  Iterations);
559
560                  ulong counter = 0;
561
562                  //var firstLink = links.First();
563                  // Создаём одну связь, из которой будет производить считывание
564                  var firstLink = links.Create();
565
566                  var sw = Stopwatch.StartNew();
567
568                  // Тестируем саму функцию
569                  for (ulong i = 0; i < Iterations; i++)
570                  {
571                      counter += links.GetSource(firstLink);
572                  }
573
574                  var elapsedTime = sw.Elapsed;
575
576                  var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
577
578                  // Удаляем связь, из которой производилось считывание
579                  links.Delete(firstLink);
580
581                  ConsoleHelpers.Debug(
582                      "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                         ↪  second), counter result: {3}",
583                      Iterations, elapsedTime, (long)iterationsPerSecond, counter);
584              }
585          }
586
587          [Fact(Skip = "performance test")]
```

```
588         public static void GetSourceInParallel()
589         {
590             using (var scope = new TempLinksTestScope())
591             {
592                 var links = scope.Links;
593                 ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
                    ↪  parallel.", Iterations);
594 
595                 long counter = 0;
596 
597                 //var firstLink = links.First();
598                 var firstLink = links.Create();
599 
600                 var sw = Stopwatch.StartNew();
601 
602                 // Тестируем саму функцию
603                 Parallel.For(0, Iterations, x =>
604                 {
605                     Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
606                     //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: 0(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}",
753                      iterations, elapsedTime, (long)iterationsPerSecond, counter);
754             }
755         }
756
757         [Fact(Skip = "useless: 0(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         */
```

```
        /*
        [Fact]
        public static void TestParallelForeach()
        {
            var tempFilename = Path.GetTempFileName();

            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename, DefaultLinksSizeStep))
            {

                long counter = 0;

                ConsoleHelpers.Debug("Testing parallel foreach through links.");

                var sw = Stopwatch.StartNew();

                //Parallel.ForEach((IEnumerable<ulong>)links, x =>
                //{
                //    Interlocked.Increment(ref counter);
                //});

                var elapsedTime = sw.Elapsed;

                var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
            }

            File.Delete(tempFilename);
        }
        */

        [Fact(Skip = "performance test")]
        public static void Create64BillionLinks()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var linksBeforeTest = links.Count();

                long linksToCreate = 64 * 1024 * 1024 / UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;

                ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);

                var elapsedTime = Performance.Measure(() =>
                {
                    for (long i = 0; i < linksToCreate; i++)
                    {
                        links.Create();
                    }
                });

                var linksCreated = links.Count() - linksBeforeTest;
                var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("Current links count: {0}.", links.Count());

                ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    linksCreated, elapsedTime,
                    (long)linksPerSecond);
            }
        }

        [Fact(Skip = "performance test")]
        public static void Create64BillionLinksInParallel()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var linksBeforeTest = links.Count();

                var sw = Stopwatch.StartNew();

                long linksToCreate = 64 * 1024 * 1024 / UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;

```

```
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: 0(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]));
```

```
                    Assert.Equal(numbers[i],
                    ↪  fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
                }
            }
        }
    }
}
```

./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs

```
using Xunit;
using Platform.Interfaces;
using Platform.Memory;
using Platform.Reflection;
using Platform.Scopes;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Numbers.Raw;
using Platform.Data.Doublets.Numbers.Unary;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.ResizableDirectMemory;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Unicode;

namespace Platform.Data.Doublets.Tests
{
    public static class UnicodeConvertersTests
    {
        [Fact]
        public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var meaningRoot = links.CreatePoint();
                var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
                var powerOf2ToUnaryNumberConverter = new
                ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
                var addressToUnaryNumberConverter = new
                ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var unaryNumberToAddressConverter = new
                ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                ↪  addressToUnaryNumberConverter, unaryNumberToAddressConverter);
            }
        }

        [Fact]
        public static void CharAndRawNumberUnicodeSymbolConvertersTest()
        {
            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);
        }
```

```
59
60          [Fact]
61          public static void StringAndUnicodeSequenceConvertersTest()
62          {
63              using (var scope = new TempLinksTestScope())
64              {
65                  var links = scope.Links;
66
67                  var itself = links.Constants.Itself;
68
69                  var meaningRoot = links.CreatePoint();
70                  var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
71                  var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
72                  var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
73                  var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
74                  var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
75
76                  var powerOf2ToUnaryNumberConverter = new
                    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
77                  var addressToUnaryNumberConverter = new
                    ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
78                  var charToUnicodeSymbolConverter = new
                    ↪  CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                    ↪  unicodeSymbolMarker);
79
80                  var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
81                  var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
82                  var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
83                  var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
84                  var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
85                  var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
86                  var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
87                  var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
88
89                  var stringToUnicodeSequenceConverter = new
                    ↪  StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
                    ↪  index, optimalVariantConverter, unicodeSequenceMarker);
90
91                  var originalString = "Hello";
92
93                  var unicodeSequenceLink =
                    ↪  stringToUnicodeSequenceConverter.Convert(originalString);
94
95                  var unicodeSymbolCriterionMatcher = new
                    ↪  UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
96                  var unicodeSymbolToCharConverter = new
                    ↪  UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
                    ↪  unicodeSymbolCriterionMatcher);
97
98                  var unicodeSequenceCriterionMatcher = new
                    ↪  UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
99
100                 var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
                    ↪  unicodeSymbolCriterionMatcher.IsMatched);
101
102                 var unicodeSequenceToStringConverter = new
                    ↪  UnicodeSequenceToStringConverter<ulong>(links,
                    ↪  unicodeSequenceCriterionMatcher, sequenceWalker,
                    ↪  unicodeSymbolToCharConverter);
103
104                 var resultingString =
                    ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
105
106                 Assert.Equal(originalString, resultingString);
107             }
108         }
109     }
110 }
```

Index