1.1 ./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);
        }
    }
}
```

1.2 ./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);
        }
    }
}
```

1.3 ./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);
    }
}
```

1.4 ./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();
            }
        }
    }
}
```

1.5 ./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);
        }
    }
}
```

1.6 ./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));
    }
}
```

1.7 ./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);
        }
    }
}
```

1.8 ./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)
```

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

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

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

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

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

1.11 ./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);
        }
    }
}
```

1.12 ./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);
        }
    }
}
```

1.13 ./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  }
```

1.14 ./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.Lists;
6   using Platform.Data.Universal;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets.Decorators
11  {
12      /// <remarks>
13      /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
14      /// 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).
15      ///
16      /// 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.)
17      /// </remarks>
```

```
18      internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
19      {
20          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
21
22          public UniLinks(ILinks<TLink> links) : base(links) { }
23
24          private struct Transition
25          {
26              public IList<TLink> Before;
27              public IList<TLink> After;
28
29              public Transition(IList<TLink> before, IList<TLink> after)
30              {
31                  Before = before;
32                  After = after;
33              }
34          }
35
36          //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
37          //public static readonly IReadOnlyList<TLink> NullLink = new
            ↪ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
            ↪ });
38
39          // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
            ↪ (Links-Expression)
40          public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
            ↪ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            ↪ substitutedHandler)
41          {
42              ////List<Transition> transitions = null;
43              ////if (!restriction.IsNullOrEmpty())
44              ////{
45              ////    // Есть причина делать проход (чтение)
46              ////    if (matchedHandler != null)
47              ////    {
48              ////        if (!substitution.IsNullOrEmpty())
49              ////        {
50              ////            // restriction => { 0, 0, 0 } | { 0 } // Create
51              ////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
                ↪ Create / Update
52              ////            // substitution => { 0, 0, 0 } | { 0 } // Delete
53              ////            transitions = new List<Transition>();
54              ////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
55              ////            {
56              ////                // If index is Null, that means we always ignore every other
                ↪ value (they are also Null by definition)
57              ////                var matchDecision = matchedHandler(, NullLink);
58              ////                if (Equals(matchDecision, Constants.Break))
59              ////                    return false;
60              ////                if (!Equals(matchDecision, Constants.Skip))
61              ////                    transitions.Add(new Transition(matchedLink, newValue));
62              ////            }
63              ////            else
64              ////            {
65              ////                Func<T, bool> handler;
66              ////                handler = link =>
67              ////                {
68              ////                    var matchedLink = Memory.GetLinkValue(link);
69              ////                    var newValue = Memory.GetLinkValue(link);
70              ////                    newValue[Constants.IndexPart] = Constants.Itself;
71              ////                    newValue[Constants.SourcePart] =
                ↪ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
                ↪ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
72              ////                    newValue[Constants.TargetPart] =
                ↪ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
                ↪ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
73              ////                    var matchDecision = matchedHandler(matchedLink, newValue);
74              ////                    if (Equals(matchDecision, Constants.Break))
75              ////                        return false;
76              ////                    if (!Equals(matchDecision, Constants.Skip))
77              ////                        transitions.Add(new Transition(matchedLink, newValue));
78              ////                    return true;
79              ////                };
80              ////                if (!Memory.Each(handler, restriction))
81              ////                    return Constants.Break;
82              ////            }
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             //        if (!Memory.Each(handler, restriction))
162             //            return Constants.Break;
163             //    }
164             //    if (!matchedLinks.IsNullOrEmpty())
165             //    {
166             //        var totalMatchedLinks = matchedLinks.Count;
167             //        for (var i = 0; i < totalMatchedLinks; i++)
168             //        {
169             //            var matchedLink = matchedLinks[i];
170             //            if (substitutedHandler != null)
171             //            {
172             //                var newValue = new List<T>(); // TODO: Prepare value to update here
173             //                // TODO: Decide is it actually needed to use Before and After
                ↪  substitution handling.
174             //                var substitutedDecision = substitutedHandler(matchedLink,
                ↪  newValue);
175             //                if (Equals(substitutedDecision, Constants.Break))
176             //                    return Constants.Break;
177             //                if (Equals(substitutedDecision, Constants.Continue))
178             //                {
179             //                    // Actual update here
180             //                    Memory.SetLinkValue(newValue);
181             //                }
182             //                if (Equals(substitutedDecision, Constants.Skip))
183             //                {
184             //                    // Cancel the update. TODO: decide use separate Cancel
                ↪  constant or Skip is enough?
185             //                }
186             //            }
187             //        }
188             //    }
189             //}
190             return Constants.Continue;
191         }
192 
193         public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
            ↪  matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            ↪  substitutionHandler)
194         {
195             if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
196             {
197                 return Constants.Continue;
198             }
199             else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
                ↪  Check if it is a correct condition
200             {
201                 // Or it only applies to trigger without matchHandler.
202                 throw new NotImplementedException();
203             }
204             else if (!substitution.IsNullOrEmpty()) // Creation
205             {
206                 var before = Array.Empty<TLink>();
207                 // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
                    ↪  (пройти мимо) или пустить (взять)?
208                 if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  Constants.Break))
209                 {
210                     return Constants.Break;
211                 }
212                 var after = (IList<TLink>)substitution.ToArray();
213                 if (_equalityComparer.Equals(after[0], default))
214                 {
215                     var newLink = Links.Create();
216                     after[0] = newLink;
217                 }
218                 if (substitution.Count == 1)
219                 {
220                     after = Links.GetLink(substitution[0]);
221                 }
222                 else if (substitution.Count == 3)
223                 {
224                     //Links.Create(after);
225                 }
226                 else
227                 {
228                     throw new NotSupportedException();
229                 }
230                 if (matchHandler != null)
```

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

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

1.15 ./Platform.Data.Doublets/Doublet.cs

```
1   using System;
2   using System.Collections.Generic;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets
7   {
8       public struct Doublet<T> : IEquatable<Doublet<T>>
9       {
10          private static readonly EqualityComparer<T> _equalityComparer =
                ↪  EqualityComparer<T>.Default;
11
12          public T Source { get; set; }
13          public T Target { get; set; }
14
15          public Doublet(T source, T target)
16          {
17              Source = source;
18              Target = target;
19          }
20
21          public override string ToString() => $"{Source}->{Target}";
22
23          public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
                ↪  && _equalityComparer.Equals(Target, other.Target);
24
25          public override bool Equals(object obj) => obj is Doublet<T> doublet ?
                ↪  base.Equals(doublet) : false;
26
27          public override int GetHashCode() => (Source, Target).GetHashCode();
28
29          public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);
30
31          public static bool operator !=(Doublet<T> left, Doublet<T> right) => !(left == right);
32      }
33  }
```

1.16 ./Platform.Data.Doublets/DoubletComparer.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets
7   {
8       /// <remarks>
9       /// TODO: Может стоит попробовать ref во всех методах (IRefEqualityComparer)
10      /// 2x faster with comparer
11      /// </remarks>
12      public class DoubletComparer<T> : IEqualityComparer<Doublet<T>>
13      {
14          public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();
15
```

```
        [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();
    }
}
```

1.17 ./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>>
    {
    }
}
```

1.18 ./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.GetOrCreate(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;
                }
            }
```

```
55          }
56
57          public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
            ↪ links.Delete(new LinkAddress<TLink>(linkToDelete));
58
59          /// <remarks>
60          /// TODO: Возможно есть очень простой способ это сделать.
61          /// (Например просто удалить файл, или изменить его размер таким образом,
62          /// чтобы удалился весь контент)
63          /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
64          /// </remarks>
65          public static void DeleteAll<TLink>(this ILinks<TLink> links)
66          {
67              var equalityComparer = EqualityComparer<TLink>.Default;
68              var comparer = Comparer<TLink>.Default;
69              for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
                ↪ Arithmetic.Decrement(i))
70              {
71                  links.Delete(i);
72                  if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
73                  {
74                      i = links.Count();
75                  }
76              }
77          }
78
79          public static TLink First<TLink>(this ILinks<TLink> links)
80          {
81              TLink firstLink = default;
82              var equalityComparer = EqualityComparer<TLink>.Default;
83              if (equalityComparer.Equals(links.Count(), default))
84              {
85                  throw new InvalidOperationException("В хранилище нет связей.");
86              }
87              links.Each(links.Constants.Any, links.Constants.Any, link =>
88              {
89                  firstLink = link[links.Constants.IndexPart];
90                  return links.Constants.Break;
91              });
92              if (equalityComparer.Equals(firstLink, default))
93              {
94                  throw new InvalidOperationException("В процессе поиска по хранилищу не было
                    ↪ найдено связей.");
95              }
96              return firstLink;
97          }
98
99          #region Paths
100
101         /// <remarks>
102         /// TODO: Как так? Как то что ниже может быть корректно?
103         /// Скорее всего практически не применимо
104         /// Предполагалось, что можно было конвертировать формируемый в проходе через
            ↪ SequenceWalker
105         /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
106         /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
107         /// </remarks>
108         public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ path)
109         {
110             var current = path[0];
111             //EnsureLinkExists(current, "path");
112             if (!links.Exists(current))
113             {
114                 return false;
115             }
116             var equalityComparer = EqualityComparer<TLink>.Default;
117             var constants = links.Constants;
118             for (var i = 1; i < path.Length; i++)
119             {
120                 var next = path[i];
121                 var values = links.GetLink(current);
122                 var source = values[constants.SourcePart];
123                 var target = values[constants.TargetPart];
124                 if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                    ↪ next))
125                 {
126                     //throw new InvalidOperationException(string.Format("Невозможно выбрать
                        ↪ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
```

```
127                     return false;
128                 }
129                 if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                    ↪ target))
130                 {
131                     //throw new InvalidOperationException(string.Format("Невозможно продолжить
                        ↪ путь через элемент пути {0}", next));
132                     return false;
133                 }
134                 current = next;
135             }
136             return true;
137         }
138 
139         /// <remarks>
140         /// Может потребовать дополнительного стека для PathElement's при использовании
            ↪ SequenceWalker.
141         /// </remarks>
142         public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
            ↪ path)
143         {
144             links.EnsureLinkExists(root, "root");
145             var currentLink = root;
146             for (var i = 0; i < path.Length; i++)
147             {
148                 currentLink = links.GetLink(currentLink)[path[i]];
149             }
150             return currentLink;
151         }
152 
153         public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
            ↪ links, TLink root, ulong size, ulong index)
154         {
155             var constants = links.Constants;
156             var source = constants.SourcePart;
157             var target = constants.TargetPart;
158             if (!Platform.Numbers.Math.IsPowerOfTwo(size))
159             {
160                 throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
                    ↪ than powers of two are not supported.");
161             }
162             var path = new BitArray(BitConverter.GetBytes(index));
163             var length = Bit.GetLowestPosition(size);
164             links.EnsureLinkExists(root, "root");
165             var currentLink = root;
166             for (var i = length - 1; i >= 0; i--)
167             {
168                 currentLink = links.GetLink(currentLink)[path[i] ? target : source];
169             }
170             return currentLink;
171         }
172 
173         #endregion
174 
175         /// <summary>
176         /// Возвращает индекс указанной связи.
177         /// </summary>
178         /// <param name="links">Хранилище связей.</param>
179         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
180         /// <returns>Индекс начальной связи для указанной связи.</returns>
181         [MethodImpl(MethodImplOptions.AggressiveInlining)]
182         public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.IndexPart];
183 
184         /// <summary>
185         /// Возвращает индекс начальной (Source) связи для указанной связи.
186         /// </summary>
187         /// <param name="links">Хранилище связей.</param>
188         /// <param name="link">Индекс связи.</param>
189         /// <returns>Индекс начальной связи для указанной связи.</returns>
190         [MethodImpl(MethodImplOptions.AggressiveInlining)]
191         public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ links.GetLink(link)[links.Constants.SourcePart];
192 
193         /// <summary>
194         /// Возвращает индекс начальной (Source) связи для указанной связи.
195         /// </summary>
```

```
196         /// <param name="links">Хранилище связей.</param>
197         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
198         /// <returns>Индекс начальной связи для указанной связи.</returns>
199         [MethodImpl(MethodImplOptions.AggressiveInlining)]
200         public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.SourcePart];
201
202         /// <summary>
203         /// Возвращает индекс конечной (Target) связи для указанной связи.
204         /// </summary>
205         /// <param name="links">Хранилище связей.</param>
206         /// <param name="link">Индекс связи.</param>
207         /// <returns>Индекс конечной связи для указанной связи.</returns>
208         [MethodImpl(MethodImplOptions.AggressiveInlining)]
209         public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ links.GetLink(link)[links.Constants.TargetPart];
210
211         /// <summary>
212         /// Возвращает индекс конечной (Target) связи для указанной связи.
213         /// </summary>
214         /// <param name="links">Хранилище связей.</param>
215         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
216         /// <returns>Индекс конечной связи для указанной связи.</returns>
217         [MethodImpl(MethodImplOptions.AggressiveInlining)]
218         public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.TargetPart];
219
220         /// <summary>
221         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
222         /// </summary>
223         /// <param name="links">Хранилище связей.</param>
224         /// <param name="handler">Обработчик каждой подходящей связи.</param>
225         /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
            ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
            ↪ Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
226         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
227         [MethodImpl(MethodImplOptions.AggressiveInlining)]
228         public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
            ↪ handler, params TLink[] restrictions)
229             => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
                ↪ links.Constants.Continue);
230
231         /// <summary>
232         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
233         /// </summary>
234         /// <param name="links">Хранилище связей.</param>
235         /// <param name="source">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
            ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
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 EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
            ↪ restrictions)
302         {
303             for (var i = 0; i < restrictions.Count; i++)
304             {
305                 if (!links.Exists(restrictions[i]))
306                 {
307                     throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                        ↪ $"sequence[{i}]");
308                 }
309             }
310         }
311
312         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
313         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
            ↪ reference, string argumentName)
314         {
315             if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
316             {
317                 throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
318             }
319         }
320 
321         [MethodImpl(MethodImplOptions.AggressiveInlining)]
322         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
            ↪ IList<TLink> restrictions, string argumentName)
323         {
324             for (int i = 0; i < restrictions.Count; i++)
325             {
326                 links.EnsureInnerReferenceExists(restrictions[i], argumentName);
327             }
328         }
329 
330         [MethodImpl(MethodImplOptions.AggressiveInlining)]
331         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
            ↪ restrictions)
332         {
333             var equalityComparer = EqualityComparer<TLink>.Default;
334             var any = links.Constants.Any;
335             for (var i = 0; i < restrictions.Count; i++)
336             {
337                 if (!equalityComparer.Equals(restrictions[i], any) &&
                    ↪  !links.Exists(restrictions[i]))
338                 {
339                     throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                        ↪  $"sequence[{i}]");
340                 }
341             }
342         }
343 
344         [MethodImpl(MethodImplOptions.AggressiveInlining)]
345         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
            ↪ string argumentName)
346         {
347             var equalityComparer = EqualityComparer<TLink>.Default;
348             if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
349             {
350                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
351             }
352         }
353 
354         [MethodImpl(MethodImplOptions.AggressiveInlining)]
355         public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
            ↪ link, string argumentName)
356         {
357             var equalityComparer = EqualityComparer<TLink>.Default;
358             if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
359             {
360                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
361             }
362         }
363 
364         /// <param name="links">Хранилище связей.</param>
365         [MethodImpl(MethodImplOptions.AggressiveInlining)]
366         public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
            ↪ TLink target)
367         {
368             if (links.Exists(source, target))
369             {
370                 throw new LinkWithSameValueAlreadyExistsException();
371             }
372         }
373 
374         /// <param name="links">Хранилище связей.</param>
375         public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
376         {
377             if (links.HasUsages(link))
378             {
379                 throw new ArgumentLinkHasDependenciesException<TLink>(link);
380             }
381         }
382 
```

```
383         /// <param name="links">Хранилище связей.</param>
384         public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ addresses) => links.EnsureCreated(links.Create, addresses);
385
386         /// <param name="links">Хранилище связей.</param>
387         public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
388
389         /// <param name="links">Хранилище связей.</param>
390         public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
            ↪ params TLink[] addresses)
391         {
392             var constants = links.Constants;
393             var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
                ↪ !links.Exists(x)));
394             if (nonExistentAddresses.Count > 0)
395             {
396                 var max = nonExistentAddresses.Max();
397                 max = (Integer<TLink>)System.Math.Min((ulong)(Integer<TLink>)max,
                    ↪ (ulong)(Integer<TLink>)constants.InternalReferencesRange.Maximum);
398                 var createdLinks = new List<TLink>();
399                 var equalityComparer = EqualityComparer<TLink>.Default;
400                 TLink createdLink = creator();
401                 while (!equalityComparer.Equals(createdLink, max))
402                 {
403                     createdLinks.Add(createdLink);
404                 }
405                 for (var i = 0; i < createdLinks.Count; i++)
406                 {
407                     if (!nonExistentAddresses.Contains(createdLinks[i]))
408                     {
409                         links.Delete(createdLinks[i]);
410                     }
411                 }
412             }
413         }
414
415         #endregion
416
417         /// <param name="links">Хранилище связей.</param>
418         public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
419         {
420             var constants = links.Constants;
421             var values = links.GetLink(link);
422             TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
                ↪ constants.Any));
423             var equalityComparer = EqualityComparer<TLink>.Default;
424             if (equalityComparer.Equals(values[constants.SourcePart], link))
425             {
426                 usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
427             }
428             TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
                ↪ link));
429             if (equalityComparer.Equals(values[constants.TargetPart], link))
430             {
431                 usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
432             }
433             return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
434         }
435
436         /// <param name="links">Хранилище связей.</param>
437         [MethodImpl(MethodImplOptions.AggressiveInlining)]
438         public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ Comparer<TLink>.Default.Compare(links.CountUsages(link), Integer<TLink>.Zero) > 0;
439
440         /// <param name="links">Хранилище связей.</param>
441         [MethodImpl(MethodImplOptions.AggressiveInlining)]
442         public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
            ↪ TLink target)
443         {
444             var constants = links.Constants;
445             var values = links.GetLink(link);
446             var equalityComparer = EqualityComparer<TLink>.Default;
447             return equalityComparer.Equals(values[constants.SourcePart], source) &&
                ↪ equalityComparer.Equals(values[constants.TargetPart], target);
448         }
449
450         /// <summary>
```

```
451         /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
452         /// </summary>
453         /// <param name="links">Хранилище связей.</param>
454         /// <param name="source">Индекс связи, которая является началом для искомой
            ↪  связи.</param>
455         /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
456         /// <returns>Индекс искомой связи с указанными Source (началом) и Target
            ↪  (концом).</returns>
457         [MethodImpl(MethodImplOptions.AggressiveInlining)]
458         public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪  target)
459         {
460             var contants = links.Constants;
461             var setter = new Setter<TLink, TLink>(contants.Continue, contants.Break, default);
462             links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
463             return setter.Result;
464         }
465
466         /// <param name="links">Хранилище связей.</param>
467         [MethodImpl(MethodImplOptions.AggressiveInlining)]
468         public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
469
470         /// <param name="links">Хранилище связей.</param>
471         [MethodImpl(MethodImplOptions.AggressiveInlining)]
472         public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
473         {
474             var link = links.Create();
475             return links.Update(link, link, link);
476         }
477
478         /// <param name="links">Хранилище связей.</param>
479         [MethodImpl(MethodImplOptions.AggressiveInlining)]
480         public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪  target) => links.Update(links.Create(), source, target);
481
482         /// <summary>
483         /// Обновляет связь с указанными началом (Source) и концом (Target)
484         /// на связь с указанными началом (NewSource) и концом (NewTarget).
485         /// </summary>
486         /// <param name="links">Хранилище связей.</param>
487         /// <param name="link">Индекс обновляемой связи.</param>
488         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪  выполняется обновление.</param>
489         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪  выполняется обновление.</param>
490         /// <returns>Индекс обновлённой связи.</returns>
491         [MethodImpl(MethodImplOptions.AggressiveInlining)]
492         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));
493
494         /// <summary>
495         /// Обновляет связь с указанными началом (Source) и концом (Target)
496         /// на связь с указанными началом (NewSource) и концом (NewTarget).
497         /// </summary>
498         /// <param name="links">Хранилище связей.</param>
499         /// <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>
500         /// <returns>Индекс обновлённой связи.</returns>
501         [MethodImpl(MethodImplOptions.AggressiveInlining)]
502         public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
503         {
504             if (restrictions.Length == 2)
505             {
506                 return links.MergeAndDelete(restrictions[0], restrictions[1]);
507             }
508             if (restrictions.Length == 4)
509             {
510                 return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
                    ↪  restrictions[2], restrictions[3]);
511             }
512             else
513             {
514                 return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
515             }
```

```
516         }
517
518         [MethodImpl(MethodImplOptions.AggressiveInlining)]
519         public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
            ↪  links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
520         {
521             var equalityComparer = EqualityComparer<TLink>.Default;
522             var constants = links.Constants;
523             var restrictionsIndex = restrictions[constants.IndexPart];
524             var substitutionIndex = substitution[constants.IndexPart];
525             if (equalityComparer.Equals(substitutionIndex, default))
526             {
527                 substitutionIndex = restrictionsIndex;
528             }
529             var source = substitution[constants.SourcePart];
530             var target = substitution[constants.TargetPart];
531             source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
532             target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
533             return new Link<TLink>(substitutionIndex, source, target);
534         }
535
536         /// <summary>
537         /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
            ↪  с указанными Source (началом) и Target (концом).
538         /// </summary>
539         /// <param name="links">Хранилище связей.</param>
540         /// <param name="source">Индекс связи, которая является началом на создаваемой
            ↪  связи.</param>
541         /// <param name="target">Индекс связи, которая является концом для создаваемой
            ↪  связи.</param>
542         /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
543         [MethodImpl(MethodImplOptions.AggressiveInlining)]
544         public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪  target)
545         {
546             var link = links.SearchOrDefault(source, target);
547             if (EqualityComparer<TLink>.Default.Equals(link, default))
548             {
549                 link = links.CreateAndUpdate(source, target);
550             }
551             return link;
552         }
553
554         /// <summary>
555         /// Обновляет связь с указанными началом (Source) и концом (Target)
556         /// на связь с указанными началом (NewSource) и концом (NewTarget).
557         /// </summary>
558         /// <param name="links">Хранилище связей.</param>
559         /// <param name="source">Индекс связи, которая является началом обновляемой
            ↪  связи.</param>
560         /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
561         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪  выполняется обновление.</param>
562         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪  выполняется обновление.</param>
563         /// <returns>Индекс обновлённой связи.</returns>
564         [MethodImpl(MethodImplOptions.AggressiveInlining)]
565         public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
            ↪  TLink target, TLink newSource, TLink newTarget)
566         {
567             var equalityComparer = EqualityComparer<TLink>.Default;
568             var link = links.SearchOrDefault(source, target);
569             if (equalityComparer.Equals(link, default))
570             {
571                 return links.CreateAndUpdate(newSource, newTarget);
572             }
573             if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
                ↪  target))
574             {
575                 return link;
576             }
577             return links.Update(link, newSource, newTarget);
578         }
579
580         /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
581         /// <param name="links">Хранилище связей.</param>
582         /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
```

```
583             /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
584             [MethodImpl(MethodImplOptions.AggressiveInlining)]
585             public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
                ↪ target)
586             {
587                 var link = links.SearchOrDefault(source, target);
588                 if (!EqualityComparer<TLink>.Default.Equals(link, default))
589                 {
590                     links.Delete(link);
591                     return link;
592                 }
593                 return default;
594             }
595
596             /// <summary>Удаляет несколько связей.</summary>
597             /// <param name="links">Хранилище связей.</param>
598             /// <param name="deletedLinks">Список адресов связей к удалению.</param>
599             [MethodImpl(MethodImplOptions.AggressiveInlining)]
600             public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
601             {
602                 for (int i = 0; i < deletedLinks.Count; i++)
603                 {
604                     links.Delete(deletedLinks[i]);
605                 }
606             }
607
608             /// <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>
609             public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
610             {
611                 var anyConstant = links.Constants.Any;
612                 var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
613                 links.DeleteByQuery(usagesAsSourceQuery);
614                 var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
615                 links.DeleteByQuery(usagesAsTargetQuery);
616             }
617
618             public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
619             {
620                 var count = (Integer<TLink>)links.Count(query);
621                 if (count > 0)
622                 {
623                     var queryResult = new TLink[count];
624                     var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
                        ↪ links.Constants.Continue);
625                     links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
626                     for (var i = (long)count - 1; i >= 0; i--)
627                     {
628                         links.Delete(queryResult[i]);
629                     }
630                 }
631             }
632
633             // TODO: Move to Platform.Data
634             public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
635             {
636                 var nullConstant = links.Constants.Null;
637                 var equalityComparer = EqualityComparer<TLink>.Default;
638                 var link = links.GetLink(linkIndex);
639                 for (int i = 1; i < link.Count; i++)
640                 {
641                     if (!equalityComparer.Equals(link[i], nullConstant))
642                     {
643                         return false;
644                     }
645                 }
646                 return true;
647             }
648
649             // TODO: Create a universal version of this method in Platform.Data (with using of for
                ↪ loop)
650             public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
651             {
652                 var nullConstant = links.Constants.Null;
653                 var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
654                 links.Update(updateRequest);
655             }
```

```
656
657         // TODO: Create a universal version of this method in Platform.Data (with using of for
            ↪ loop)
658         public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
659         {
660             if (!links.AreValuesReset(linkIndex))
661             {
662                 links.ResetValues(linkIndex);
663             }
664         }
665
666         /// <summary>
667         /// Merging two usages graphs, all children of old link moved to be children of new link
            ↪ or deleted.
668         /// </summary>
669         public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
            ↪ TLink newLinkIndex)
670         {
671             var equalityComparer = EqualityComparer<TLink>.Default;
672             if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
673             {
674                 var constants = links.Constants;
675                 var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
                    ↪ constants.Any);
676                 long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
677                 var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
                    ↪ oldLinkIndex);
678                 long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
679                 var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
                    ↪ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
680                 if (!isStandalonePoint)
681                 {
682                     var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
683                     if (totalUsages > 0)
684                     {
685                         var usages = ArrayPool.Allocate<TLink>(totalUsages);
686                         var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
                            ↪ links.Constants.Continue);
687                         var i = 0L;
688                         if (usagesAsSourceCount > 0)
689                         {
690                             links.Each(usagesFiller.AddFirstAndReturnConstant,
                                ↪ usagesAsSourceQuery);
691                             for (; i < usagesAsSourceCount; i++)
692                             {
693                                 var usage = usages[i];
694                                 if (!equalityComparer.Equals(usage, oldLinkIndex))
695                                 {
696                                     links.Update(usage, newLinkIndex, links.GetTarget(usage));
697                                 }
698                             }
699                         }
700                         if (usagesAsTargetCount > 0)
701                         {
702                             links.Each(usagesFiller.AddFirstAndReturnConstant,
                                ↪ usagesAsTargetQuery);
703                             for (; i < usages.Length; i++)
704                             {
705                                 var usage = usages[i];
706                                 if (!equalityComparer.Equals(usage, oldLinkIndex))
707                                 {
708                                     links.Update(usage, links.GetSource(usage), newLinkIndex);
709                                 }
710                             }
711                         }
712                         ArrayPool.Free(usages);
713                     }
714                 }
715             }
716             return newLinkIndex;
717         }
718
719         /// <summary>
720         /// Replace one link with another (replaced link is deleted, children are updated or
            ↪ deleted).
721         /// </summary>
722         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
723         public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
            ↪ TLink newLinkIndex)
724         {
725             var equalityComparer = EqualityComparer<TLink>.Default;
726             if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
727             {
728                 links.MergeUsages(oldLinkIndex, newLinkIndex);
729                 links.Delete(oldLinkIndex);
730             }
731             return newLinkIndex;
732         }
733 
734         public static ILinks<TLink>
            ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
735         {
736             links = new LinksCascadeUsagesResolver<TLink>(links);
737             links = new NonNullContentsLinkDeletionResolver<TLink>(links);
738             links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
739             return links;
740         }
741     }
742 }
```

1.19 ./Platform.Data.Doublets/ISynchronizedLinks.cs

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

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

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

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

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

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

1.22 ./Platform.Data.Doublets/Link.cs

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

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

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

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

        private const int Length = 3;

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

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

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

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

```
            SetValues(otherList, out Index, out Source, out Target);
        }
        else
        {
            throw new NotSupportedException();
        }
    }

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

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

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

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

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

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

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

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

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

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
120         public static string ToString(TLink source, TLink target) => $"({source}->{target})";
121
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         public static implicit operator TLink[](Link<TLink> link) => link.ToArray();
124
125         [MethodImpl(MethodImplOptions.AggressiveInlining)]
126         public static implicit operator Link<TLink>(TLink[] linkArray) => new
            ↪ Link<TLink>(linkArray);
127
128         [MethodImpl(MethodImplOptions.AggressiveInlining)]
129         public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
            ↪ ToString(Source, Target) : ToString(Index, Source, Target);
130
131         #region IList
132
133         public int Count => Length;
134
135         public bool IsReadOnly => true;
136
137         public TLink this[int index]
138         {
139             [MethodImpl(MethodImplOptions.AggressiveInlining)]
140             get
141             {
142                 Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
                    ↪ nameof(index));
143                 if (index == _constants.IndexPart)
144                 {
145                     return Index;
146                 }
147                 if (index == _constants.SourcePart)
148                 {
149                     return Source;
150                 }
151                 if (index == _constants.TargetPart)
152                 {
153                     return Target;
154                 }
155                 throw new NotSupportedException(); // Impossible path due to
                    ↪ Ensure.ArgumentInRange
156             }
157             [MethodImpl(MethodImplOptions.AggressiveInlining)]
158             set => throw new NotSupportedException();
159         }
160
161         [MethodImpl(MethodImplOptions.AggressiveInlining)]
162         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
163
164         [MethodImpl(MethodImplOptions.AggressiveInlining)]
165         public IEnumerator<TLink> GetEnumerator()
166         {
167             yield return Index;
168             yield return Source;
169             yield return Target;
170         }
171
172         [MethodImpl(MethodImplOptions.AggressiveInlining)]
173         public void Add(TLink item) => throw new NotSupportedException();
174
175         [MethodImpl(MethodImplOptions.AggressiveInlining)]
176         public void Clear() => throw new NotSupportedException();
177
178         [MethodImpl(MethodImplOptions.AggressiveInlining)]
179         public bool Contains(TLink item) => IndexOf(item) >= 0;
180
181         [MethodImpl(MethodImplOptions.AggressiveInlining)]
182         public void CopyTo(TLink[] array, int arrayIndex)
183         {
184             Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
185             Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                ↪ nameof(arrayIndex));
186             if (arrayIndex + Length > array.Length)
187             {
188                 throw new InvalidOperationException();
189             }
190             array[arrayIndex++] = Index;
191             array[arrayIndex++] = Source;
192             array[arrayIndex] = Target;
193         }
```

```
194
195         [MethodImpl(MethodImplOptions.AggressiveInlining)]
196         public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
197
198         [MethodImpl(MethodImplOptions.AggressiveInlining)]
199         public int IndexOf(TLink item)
200         {
201             if (_equalityComparer.Equals(Index, item))
202             {
203                 return _constants.IndexPart;
204             }
205             if (_equalityComparer.Equals(Source, item))
206             {
207                 return _constants.SourcePart;
208             }
209             if (_equalityComparer.Equals(Target, item))
210             {
211                 return _constants.TargetPart;
212             }
213             return -1;
214         }
215
216         [MethodImpl(MethodImplOptions.AggressiveInlining)]
217         public void Insert(int index, TLink item) => throw new NotSupportedException();
218
219         [MethodImpl(MethodImplOptions.AggressiveInlining)]
220         public void RemoveAt(int index) => throw new NotSupportedException();
221
222         [MethodImpl(MethodImplOptions.AggressiveInlining)]
223         public static bool operator ==(Link<TLink> left, Link<TLink> right) =>
            ↪ left.Equals(right);
224
225         [MethodImpl(MethodImplOptions.AggressiveInlining)]
226         public static bool operator !=(Link<TLink> left, Link<TLink> right) => !(left == right);
227
228         #endregion
229     }
230 }
```

1.23 ./Platform.Data.Doublets/LinkExtensions.cs

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

1.24 ./Platform.Data.Doublets/LinksOperatorBase.cs

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

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

```
1  using System.Collections.Generic;
2  using Platform.Reflection;
3  using Platform.Converters;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
10     public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
           ↪ IConverter<TLink>
11     {
```

```
        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>.BitsSize; 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;
        }
    }
}
```

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

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

#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 IProperty<TLink, TLink> _frequencyPropertyOperator;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;

        public LinkToItsFrequencyNumberConveter(
            ILinks<TLink> links,
            IProperty<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);
        }
    }
}
```

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

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

```
using Platform.Ranges;
using Platform.Converters;

#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;
        }
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Converters;
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);
            }
        }

```

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

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Reflection;
4   using Platform.Converters;
5   using Platform.Numbers;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Numbers.Unary
10  {
11      public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
        ↪ IConverter<TLink>
12      {
13          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
14
15          private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
16
17          public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
            ↪ TLink> powerOf2ToUnaryNumberConverter)
18              : base(links)
19          {
20              _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
21              for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
22              {
23                  _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
24              }
25          }
26
27          public TLink Convert(TLink sourceNumber)
28          {
29              var nullConstant = Links.Constants.Null;
30              var source = sourceNumber;
31              var target = nullConstant;
32              if (!_equalityComparer.Equals(source, nullConstant))
33              {
34                  while (true)
35                  {
36                      if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
37                      {
38                          SetBit(ref target, powerOf2Index);
```

```
                        break;
                    }
                    else
                    {
                        powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
                        SetBit(ref target, powerOf2Index);
                        source = Links.GetTarget(source);
                    }
                }
            }
            return target;
        }

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

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.PropertyOperators
{
    public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IProperty<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);
            }
        }
    }
}
```

1.32 ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksAvlBalancedTreeMethodsBase.cs

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

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

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

        protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
        ↪  byte* header)
```

```
        {
            Links = links;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
        {
            unchecked
            {
                var previousValue = storedValue;
```

```
                var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4, 1);
                storedValue = modified;
            }
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected bool IsChild(TLink parent, TLink possibleChild)
        {
            var parentSize = GetSize(parent);
            var childSize = GetSizeOrZero(possibleChild);
            return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
        }

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

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

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

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

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

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

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

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

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

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

1.33 ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSizeBalancedTreeMethodsBase.cs

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

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

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> :
    ↪ SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
    {
        protected readonly TLink Break;
        protected readonly TLink Continue;
        protected readonly byte* Links;
        protected readonly byte* Header;

```

```
20          protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
            ↪  byte* header)
21          {
22              Links = links;
23              Header = header;
24              Break = constants.Break;
25              Continue = constants.Continue;
26          }
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected abstract TLink GetTreeRoot();
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected abstract TLink GetBasePartValue(TLink link);
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
            ↪  rootSource, TLink rootTarget);
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
            ↪  rootSource, TLink rootTarget);
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
            ↪  AsRef<LinksHeader<TLink>>(Header);
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
            ↪  AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes * (Integer<TLink>)link));
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
48          {
49              ref var link = ref GetLinkReference(linkIndex);
50              return new Link<TLink>(linkIndex, link.Source, link.Target);
51          }
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
55          {
56              ref var firstLink = ref GetLinkReference(first);
57              ref var secondLink = ref GetLinkReference(second);
58              return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
59          }
60
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
63          {
64              ref var firstLink = ref GetLinkReference(first);
65              ref var secondLink = ref GetLinkReference(second);
66              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
67          }
68
69          public TLink this[TLink index]
70          {
71              get
72              {
73                  var root = GetTreeRoot();
74                  if (GreaterOrEqualThan(index, GetSize(root)))
75                  {
76                      return Zero;
77                  }
78                  while (!EqualToZero(root))
79                  {
80                      var left = GetLeftOrDefault(root);
81                      var leftSize = GetSizeOrZero(left);
82                      if (LessThan(index, leftSize))
83                      {
84                          root = left;
85                          continue;
86                      }
87                      if (AreEqual(index, leftSize))
88                      {
89                          return root;
90                      }
91                      root = GetRightOrDefault(root);
```

```
92                      index = Subtract(index, Increment(leftSize));
93                  }
94                  return Zero; // TODO: Impossible situation exception (only if tree structure
                    ↪ broken)
95              }
96          }
97
98          /// <summary>
99          /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪ (концом).
100         /// </summary>
101         /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
102         /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
103         /// <returns>Индекс искомой связи.</returns>
104         public TLink Search(TLink source, TLink target)
105         {
106             var root = GetTreeRoot();
107             while (!EqualToZero(root))
108             {
109                 ref var rootLink = ref GetLinkReference(root);
110                 var rootSource = rootLink.Source;
111                 var rootTarget = rootLink.Target;
112                 if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                    ↪ node.Key < root.Key
113                 {
114                     root = GetLeftOrDefault(root);
115                 }
116                 else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                    ↪ node.Key > root.Key
117                 {
118                     root = GetRightOrDefault(root);
119                 }
120                 else // node.Key == root.Key
121                 {
122                     return root;
123                 }
124             }
125             return Zero;
126         }
127
128         // TODO: Return indices range instead of references count
129         public TLink CountUsages(TLink link)
130         {
131             var root = GetTreeRoot();
132             var total = GetSize(root);
133             var totalRightIgnore = Zero;
134             while (!EqualToZero(root))
135             {
136                 var @base = GetBasePartValue(root);
137                 if (LessOrEqualThan(@base, link))
138                 {
139                     root = GetRightOrDefault(root);
140                 }
141                 else
142                 {
143                     totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
144                     root = GetLeftOrDefault(root);
145                 }
146             }
147             root = GetTreeRoot();
148             var totalLeftIgnore = Zero;
149             while (!EqualToZero(root))
150             {
151                 var @base = GetBasePartValue(root);
152                 if (GreaterOrEqualThan(@base, link))
153                 {
154                     root = GetLeftOrDefault(root);
155                 }
156                 else
157                 {
158                     totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
159
160                     root = GetRightOrDefault(root);
161                 }
162             }
163             return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
164         }
165
```

```
166         [MethodImpl(MethodImplOptions.AggressiveInlining)]
167         public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
            ↪ EachUsageCore(@base, GetTreeRoot(), handler);
168
169         // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
            ↪ low-level MSIL stack.
170         private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
171         {
172             var @continue = Continue;
173             if (EqualToZero(link))
174             {
175                 return @continue;
176             }
177             var linkBasePart = GetBasePartValue(link);
178             var @break = Break;
179             if (GreaterThan(linkBasePart, @base))
180             {
181                 if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
182                 {
183                     return @break;
184                 }
185             }
186             else if (LessThan(linkBasePart, @base))
187             {
188                 if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
189                 {
190                     return @break;
191                 }
192             }
193             else //if (linkBasePart == @base)
194             {
195                 if (AreEqual(handler(GetLinkValues(link)), @break))
196                 {
197                     return @break;
198                 }
199                 if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
200                 {
201                     return @break;
202                 }
203                 if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
204                 {
205                     return @break;
206                 }
207             }
208             return @continue;
209         }
210
211         protected override void PrintNodeValue(TLink node, StringBuilder sb)
212         {
213             ref var link = ref GetLinkReference(node);
214             sb.Append(' ');
215             sb.Append(link.Source);
216             sb.Append('-');
217             sb.Append('>');
218             sb.Append(link.Target);
219         }
220     }
221 }
```

1.34 ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesAvlBalancedTreeMethods.cs

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

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

1.35 ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesSizeBalancedTreeMethods.cs

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

1.36 ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsAvlBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
12          protected unsafe override ref TLink GetLeftReference(TLink node) => ref
            ↪  GetLinkReference(node).LeftAsTarget;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected unsafe override ref TLink GetRightReference(TLink node) => ref
            ↪  GetLinkReference(node).RightAsTarget;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override void SetLeft(TLink node, TLink left) =>
            ↪  GetLinkReference(node).LeftAsTarget = left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(TLink node, TLink right) =>
            ↪  GetLinkReference(node).RightAsTarget = right;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override TLink GetSize(TLink node) =>
            ↪  GetSizeValue(GetLinkReference(node).SizeAsTarget);
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
            ↪  GetLinkReference(node).SizeAsTarget, size);
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override bool GetLeftIsChild(TLink node) =>
            ↪  GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          protected override void SetLeftIsChild(TLink node, bool value) =>
            ↪  SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override bool GetRightIsChild(TLink node) =>
            ↪  GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          protected override void SetRightIsChild(TLink node, bool value) =>
            ↪  SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          protected override sbyte GetBalance(TLink node) =>
            ↪  GetBalanceValue(GetLinkReference(node).SizeAsTarget);
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
            ↪  GetLinkReference(node).SizeAsTarget, value);
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
            ↪  (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
            ↪  (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          protected override void ClearNode(TLink node)
67          {
68              ref var link = ref GetLinkReference(node);
69              link.LeftAsTarget = Zero;
70              link.RightAsTarget = Zero;
71              link.SizeAsTarget = Zero;
72          }
73      }
```

```
}
```

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
11 {
12     public unsafe partial class ResizableDirectMemoryLinks<TLink> :
       ↪ ResizableDirectMemoryLinksBase<TLink>
13     {
14         private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
15         private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
16         private byte* _header;
17         private byte* _links;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public ResizableDirectMemoryLinks(string address) : this(address, DefaultLinksSizeStep)
           ↪ { }
21
22         /// <summary>
23         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
           ↪ минимальным шагом расширения базы данных.
24         /// </summary>
25         /// <param name="address">Полный пусть к файлу базы данных.</param>
26         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
           ↪ байтах.</param>
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public ResizableDirectMemoryLinks(string address, long memoryReservationStep) : this(new
           ↪ FileMappedResizableDirectMemory(address, memoryReservationStep),
           ↪ memoryReservationStep) { }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
           ↪ DefaultLinksSizeStep) { }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
           ↪ memoryReservationStep) : this(memory, memoryReservationStep,
           ↪ Default<LinksConstants<TLink>>.Instance, true) { }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
           ↪ memoryReservationStep, LinksConstants<TLink> constants, bool useAvlBasedIndex) :
           ↪ base(memory, memoryReservationStep, constants)
38         {
39             if (useAvlBasedIndex)
40             {
41                 _createSourceTreeMethods = () => new
                   ↪ LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
42                 _createTargetTreeMethods = () => new
                   ↪ LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
43             }
44             else
45             {
46                 _createSourceTreeMethods = () => new
                   ↪ LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
47                 _createTargetTreeMethods = () => new
                   ↪ LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
48             }
49             Init(memory, memoryReservationStep);
50         }
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override void SetPointers(IResizableDirectMemory memory)
54         {
55             _links = (byte*)memory.Pointer;
56             _header = _links;
57             SourcesTreeMethods = _createSourceTreeMethods();
58             TargetsTreeMethods = _createTargetTreeMethods();
59             UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
60         }
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected override void ResetPointers()
64         {
65             base.ResetPointers();
66             _links = null;
67             _header = null;
68         }
69
70         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Disposables;
5   using Platform.Singletons;
6   using Platform.Numbers;
7   using Platform.Memory;
8   using Platform.Data.Exceptions;
9
10  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12  namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
13  {
14      public abstract class ResizableDirectMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
15      {
16          protected static readonly EqualityComparer<TLink> EqualityComparer =
            ↪ EqualityComparer<TLink>.Default;
17          protected static readonly Comparer<TLink> Comparer = Comparer<TLink>.Default;
18
19          /// <summary>Возвращает размер одной связи в байтах.</summary>
20          /// <remarks>
21          /// Используется только во вне класса, не рекомедуется использовать внутри.
22          /// Так как во вне не обязательно будет доступен unsafe C#.
23          /// </remarks>
24          public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
25
26          public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
27
28          public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
29
30          protected readonly IResizableDirectMemory _memory;
31          protected readonly long _memoryReservationStep;
32
33          protected ILinksTreeMethods<TLink> TargetsTreeMethods;
34          protected ILinksTreeMethods<TLink> SourcesTreeMethods;
35          // 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
36          protected ILinksListMethods<TLink> UnusedLinksListMethods;
37
38          /// <summary>
39          /// Возвращает общее число связей находящихся в хранилище.
40          /// </summary>
41          protected virtual TLink Total
42          {
43              get
44              {
45                  ref var header = ref GetHeaderReference();
46                  return Subtract(header.AllocatedLinks, header.FreeLinks);
47              }
48          }
49
50          public virtual LinksConstants<TLink> Constants { get; }
51
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
53          protected ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
            ↪ memoryReservationStep, LinksConstants<TLink> constants)
54          {
55              _memory = memory;
56              _memoryReservationStep = memoryReservationStep;
57              Constants = constants;
58          }
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
            ↪ memoryReservationStep) : this(memory, memoryReservationStep,
            ↪ Default<LinksConstants<TLink>>.Instance) { }
62
63          protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
64          {
65              if (memory.ReservedCapacity < memoryReservationStep)
```

```
66              {
67                  memory.ReservedCapacity = memoryReservationStep;
68              }
69              SetPointers(_memory);
70              ref var header = ref GetHeaderReference();
71              // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
72              _memory.UsedCapacity = (ConvertToUInt64(header.AllocatedLinks) * LinkSizeInBytes) +
                ↪  LinkHeaderSizeInBytes;
73              // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
74              header.ReservedLinks = ConvertToAddress((_memory.ReservedCapacity -
                ↪  LinkHeaderSizeInBytes) / LinkSizeInBytes);
75          }
76
77          [MethodImpl(MethodImplOptions.AggressiveInlining)]
78          public virtual TLink Count(IList<TLink> restrictions)
79          {
80              // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
81              if (restrictions.Count == 0)
82              {
83                  return Total;
84              }
85              var constants = Constants;
86              var any = constants.Any;
87              var index = restrictions[constants.IndexPart];
88              if (restrictions.Count == 1)
89              {
90                  if (AreEqual(index, any))
91                  {
92                      return Total;
93                  }
94                  return Exists(index) ? GetOne() : GetZero();
95              }
96              if (restrictions.Count == 2)
97              {
98                  var value = restrictions[1];
99                  if (AreEqual(index, any))
100                 {
101                     if (AreEqual(value, any))
102                     {
103                         return Total; // Any - как отсутствие ограничения
104                     }
105                     return Add(SourcesTreeMethods.CountUsages(value),
                        ↪  TargetsTreeMethods.CountUsages(value));
106                 }
107                 else
108                 {
109                     if (!Exists(index))
110                     {
111                         return GetZero();
112                     }
113                     if (AreEqual(value, any))
114                     {
115                         return GetOne();
116                     }
117                     ref var storedLinkValue = ref GetLinkReference(index);
118                     if (AreEqual(storedLinkValue.Source, value) ||
                        ↪  AreEqual(storedLinkValue.Target, value))
119                     {
120                         return GetOne();
121                     }
122                     return GetZero();
123                 }
124             }
125             if (restrictions.Count == 3)
126             {
127                 var source = restrictions[constants.SourcePart];
128                 var target = restrictions[constants.TargetPart];
129                 if (AreEqual(index, any))
130                 {
131                     if (AreEqual(source, any) && AreEqual(target, any))
132                     {
133                         return Total;
134                     }
135                     else if (AreEqual(source, any))
136                     {
137                         return TargetsTreeMethods.CountUsages(target);
138                     }
139                     else if (AreEqual(target, any))
```

```
                    {
                        return SourcesTreeMethods.CountUsages(source);
                    }
                    else //if(source != Any && target != Any)
                    {
                        // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
                        var link = SourcesTreeMethods.Search(source, target);
                        return AreEqual(link, constants.Null) ? GetZero() : GetOne();
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return GetZero();
                    }
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return GetOne();
                    }
                    ref var storedLinkValue = ref GetLinkReference(index);
                    if (!AreEqual(source, any) && !AreEqual(target, any))
                    {
                        if (AreEqual(storedLinkValue.Source, source) &&
                        ↪  AreEqual(storedLinkValue.Target, target))
                        {
                            return GetOne();
                        }
                        return GetZero();
                    }
                    var value = default(TLink);
                    if (AreEqual(source, any))
                    {
                        value = target;
                    }
                    if (AreEqual(target, any))
                    {
                        value = source;
                    }
                    if (AreEqual(storedLinkValue.Source, value) ||
                    ↪  AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪  поддерживаются.");
        }

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

```
214                 {
215                     return @continue;
216                 }
217                 return handler(GetLinkStruct(index));
218             }
219             if (restrictions.Count == 2)
220             {
221                 var value = restrictions[1];
222                 if (AreEqual(index, any))
223                 {
224                     if (AreEqual(value, any))
225                     {
226                         return Each(handler, GetEmptyList());
227                     }
228                     if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
229                     {
230                         return @break;
231                     }
232                     return Each(handler, new Link<TLink>(index, any, value));
233                 }
234                 else
235                 {
236                     if (!Exists(index))
237                     {
238                         return @continue;
239                     }
240                     if (AreEqual(value, any))
241                     {
242                         return handler(GetLinkStruct(index));
243                     }
244                     ref var storedLinkValue = ref GetLinkReference(index);
245                     if (AreEqual(storedLinkValue.Source, value) ||
246                         AreEqual(storedLinkValue.Target, value))
247                     {
248                         return handler(GetLinkStruct(index));
249                     }
250                     return @continue;
251                 }
252             }
253             if (restrictions.Count == 3)
254             {
255                 var source = restrictions[constants.SourcePart];
256                 var target = restrictions[constants.TargetPart];
257                 if (AreEqual(index, any))
258                 {
259                     if (AreEqual(source, any) && AreEqual(target, any))
260                     {
261                         return Each(handler, GetEmptyList());
262                     }
263                     else if (AreEqual(source, any))
264                     {
265                         return TargetsTreeMethods.EachUsage(target, handler);
266                     }
267                     else if (AreEqual(target, any))
268                     {
269                         return SourcesTreeMethods.EachUsage(source, handler);
270                     }
271                     else //if(source != Any && target != Any)
272                     {
273                         var link = SourcesTreeMethods.Search(source, target);
274                         return AreEqual(link, constants.Null) ? @continue :
                            ↪  handler(GetLinkStruct(link));
275                     }
276                 }
277                 else
278                 {
279                     if (!Exists(index))
280                     {
281                         return @continue;
282                     }
283                     if (AreEqual(source, any) && AreEqual(target, any))
284                     {
285                         return handler(GetLinkStruct(index));
286                     }
287                     ref var storedLinkValue = ref GetLinkReference(index);
288                     if (!AreEqual(source, any) && !AreEqual(target, any))
289                     {
290                         if (AreEqual(storedLinkValue.Source, source) &&
```

```
                        AreEqual(storedLinkValue.Target, target))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @continue;
                }
                var value = default(TLink);
                if (AreEqual(source, any))
                {
                    value = target;
                }
                if (AreEqual(target, any))
                {
                    value = source;
                }
                if (AreEqual(storedLinkValue.Source, value) ||
                    AreEqual(storedLinkValue.Target, value))
                {
                    return handler(GetLinkStruct(index));
                }
                return @continue;
            }
        }
        throw new NotSupportedException("Другие размеры и способы ограничений не
        ↪  поддерживаются.");
    }

    /// <remarks>
    /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↪  в другом месте (но не в менеджере памяти, а в логике Links)
    /// </remarks>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
    {
        var constants = Constants;
        var @null = constants.Null;
        var linkIndex = restrictions[constants.IndexPart];
        ref var link = ref GetLinkReference(linkIndex);
        ref var header = ref GetHeaderReference();
        ref var firstAsSource = ref header.FirstAsSource;
        ref var firstAsTarget = ref header.FirstAsTarget;
        // Будет корректно работать только в том случае, если пространство выделенной связи
        ↪  предварительно заполнено нулями
        if (!AreEqual(link.Source, @null))
        {
            SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
        }
        if (!AreEqual(link.Target, @null))
        {
            TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
        }
        link.Source = substitution[constants.SourcePart];
        link.Target = substitution[constants.TargetPart];
        if (!AreEqual(link.Source, @null))
        {
            SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
        }
        if (!AreEqual(link.Target, @null))
        {
            TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
        }
        return linkIndex;
    }

    /// <remarks>
    /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
    ↪  пространство
    /// </remarks>
    public virtual TLink Create(IList<TLink> restrictions)
    {
        ref var header = ref GetHeaderReference();
        var freeLink = header.FirstFreeLink;
        if (!AreEqual(freeLink, Constants.Null))
        {
            UnusedLinksListMethods.Detach(freeLink);
        }
        else
        {
            var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
```

```
366                 if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
367                 {
368                     throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
369                 }
370                 if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
371                 {
372                     _memory.ReservedCapacity += _memoryReservationStep;
373                     SetPointers(_memory);
374                     header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
                        ↪  LinkSizeInBytes);
375                 }
376                 header.AllocatedLinks = Increment(header.AllocatedLinks);
377                 _memory.UsedCapacity += LinkSizeInBytes;
378                 freeLink = header.AllocatedLinks;
379             }
380             return freeLink;
381         }
382 
383         [MethodImpl(MethodImplOptions.AggressiveInlining)]
384         public virtual void Delete(IList<TLink> restrictions)
385         {
386             ref var header = ref GetHeaderReference();
387             var link = restrictions[Constants.IndexPart];
388             if (LessThan(link, header.AllocatedLinks))
389             {
390                 UnusedLinksListMethods.AttachAsFirst(link);
391             }
392             else if (AreEqual(link, header.AllocatedLinks))
393             {
394                 header.AllocatedLinks = Decrement(header.AllocatedLinks);
395                 _memory.UsedCapacity -= LinkSizeInBytes;
396                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪  пока не дойдём до первой существующей связи
397                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
398                 while (GreaterThan(header.AllocatedLinks, GetZero()) &&
                    ↪  IsUnusedLink(header.AllocatedLinks))
399                 {
400                     UnusedLinksListMethods.Detach(header.AllocatedLinks);
401                     header.AllocatedLinks = Decrement(header.AllocatedLinks);
402                     _memory.UsedCapacity -= LinkSizeInBytes;
403                 }
404             }
405         }
406 
407         [MethodImpl(MethodImplOptions.AggressiveInlining)]
408         public IList<TLink> GetLinkStruct(TLink linkIndex)
409         {
410             ref var link = ref GetLinkReference(linkIndex);
411             return new Link<TLink>(linkIndex, link.Source, link.Target);
412         }
413 
414         /// <remarks>
415         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
            ↪  адрес реально поменялся
416         ///
417         /// Указатель this.links может быть в том же месте,
418         /// так как 0-я связь не используется и имеет такой же размер как Header,
419         /// поэтому header размещается в том же месте, что и 0-я связь
420         /// </remarks>
421         [MethodImpl(MethodImplOptions.AggressiveInlining)]
422         protected abstract void SetPointers(IResizableDirectMemory memory);
423 
424         [MethodImpl(MethodImplOptions.AggressiveInlining)]
425         protected virtual void ResetPointers()
426         {
427             SourcesTreeMethods = null;
428             TargetsTreeMethods = null;
429             UnusedLinksListMethods = null;
430         }
431 
432         [MethodImpl(MethodImplOptions.AggressiveInlining)]
433         protected abstract ref LinksHeader<TLink> GetHeaderReference();
434 
435         [MethodImpl(MethodImplOptions.AggressiveInlining)]
436         protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
437 
438         [MethodImpl(MethodImplOptions.AggressiveInlining)]
439         protected virtual bool Exists(TLink link)
440             => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
```

```
441             && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
442             && !IsUnusedLink(link);
443 
444         [MethodImpl(MethodImplOptions.AggressiveInlining)]
445         protected virtual bool IsUnusedLink(TLink linkIndex)
446         {
447             if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
                ↪  is not needed
448             {
449                 ref var link = ref GetLinkReference(linkIndex);
450                 return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
451             }
452             else
453             {
454                 return true;
455             }
456         }
457 
458         [MethodImpl(MethodImplOptions.AggressiveInlining)]
459         protected virtual TLink GetOne() => Integer<TLink>.One;
460 
461         [MethodImpl(MethodImplOptions.AggressiveInlining)]
462         protected virtual TLink GetZero() => Integer<TLink>.Zero;
463 
464         [MethodImpl(MethodImplOptions.AggressiveInlining)]
465         protected virtual bool AreEqual(TLink first, TLink second) =>
                ↪  EqualityComparer.Equals(first, second);
466 
467         [MethodImpl(MethodImplOptions.AggressiveInlining)]
468         protected virtual bool LessThan(TLink first, TLink second) => Comparer.Compare(first,
                ↪  second) < 0;
469 
470         [MethodImpl(MethodImplOptions.AggressiveInlining)]
471         protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
                ↪  Comparer.Compare(first, second) <= 0;
472 
473         [MethodImpl(MethodImplOptions.AggressiveInlining)]
474         protected virtual bool GreaterThan(TLink first, TLink second) => Comparer.Compare(first,
                ↪  second) > 0;
475 
476         [MethodImpl(MethodImplOptions.AggressiveInlining)]
477         protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
                ↪  Comparer.Compare(first, second) >= 0;
478 
479         [MethodImpl(MethodImplOptions.AggressiveInlining)]
480         protected virtual long ConvertToUInt64(TLink value) => (Integer<TLink>)value;
481 
482         [MethodImpl(MethodImplOptions.AggressiveInlining)]
483         protected virtual TLink ConvertToAddress(long value) => (Integer<TLink>)value;
484 
485         [MethodImpl(MethodImplOptions.AggressiveInlining)]
486         protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
                ↪  second);
487 
488         [MethodImpl(MethodImplOptions.AggressiveInlining)]
489         protected virtual TLink Subtract(TLink first, TLink second) =>
                ↪  Arithmetic<TLink>.Subtract(first, second);
490 
491         [MethodImpl(MethodImplOptions.AggressiveInlining)]
492         protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
493 
494         [MethodImpl(MethodImplOptions.AggressiveInlining)]
495         protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
496 
497         [MethodImpl(MethodImplOptions.AggressiveInlining)]
498         protected virtual IList<TLink> GetEmptyList() => Array.Empty<TLink>();
499 
500         #region Disposable
501 
502         protected override bool AllowMultipleDisposeCalls => true;
503 
504         protected override void Dispose(bool manual, bool wasDisposed)
505         {
506             if (!wasDisposed)
507             {
508                 ResetPointers();
509                 _memory.DisposeIfPossible();
510             }
511         }
```

```
512
513         #endregion
514     }
515 }
```

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

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

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

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets.ResizableDirectMemory
4  {
5      public interface ILinksListMethods<TLink>
6      {
```

```
        void Detach(TLink freeLink);
        void AttachAsFirst(TLink link);
    }
}
```

1.42 ./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);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public struct LinksHeader<TLink> : IEquatable<LinksHeader<TLink>>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer = _equalityComparer;

        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;

        public override bool Equals(object obj) => obj is LinksHeader<TLink> linksHeader ?
        ↪  Equals(linksHeader) : false;

        public bool Equals(LinksHeader<TLink> other)
            => _equalityComparer.Equals(AllocatedLinks, other.AllocatedLinks)
            && _equalityComparer.Equals(ReservedLinks, other.ReservedLinks)
            && _equalityComparer.Equals(FreeLinks, other.FreeLinks)
            && _equalityComparer.Equals(FirstFreeLink, other.FirstFreeLink)
            && _equalityComparer.Equals(FirstAsSource, other.FirstAsSource)
            && _equalityComparer.Equals(FirstAsTarget, other.FirstAsTarget)
            && _equalityComparer.Equals(LastFreeLink, other.LastFreeLink)
            && _equalityComparer.Equals(Reserved8, other.Reserved8);

        public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
        ↪  FirstFreeLink, FirstAsSource, FirstAsTarget, LastFreeLink, Reserved8).GetHashCode();

        public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
        ↪  left.Equals(right);

        public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
        ↪  !(left == right);
    }
}
```

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

```
using Platform.Unsafe;
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 struct RawLink<TLink> : IEquatable<RawLink<TLink>>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        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;

        public override bool Equals(object obj) => obj is RawLink<TLink> link ? Equals(link) :
        ↪  false;

        public bool Equals(RawLink<TLink> other)
            => _equalityComparer.Equals(Source, other.Source)
            && _equalityComparer.Equals(Target, other.Target)
            && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
            && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
            && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
            && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
            && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
            && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);

        public override int GetHashCode() => (Source, Target, LeftAsSource, RightAsSource,
        ↪  SizeAsSource, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();

        public static bool operator ==(RawLink<TLink> left, RawLink<TLink> right) =>
        ↪  left.Equals(right);

        public static bool operator !=(RawLink<TLink> left, RawLink<TLink> right) => !(left ==
        ↪  right);
    }
}
```

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

```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.ResizableDirectMemory.Generic;
using static System.Runtime.CompilerServices.Unsafe;

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

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

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

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

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

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

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

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

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

```
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
            ↪  always true for ulong
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
            ↪  always >= 0 for ulong
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
            ↪  for ulong
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override bool LessThan(ulong first, ulong second) => first < second;
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected override ulong Increment(ulong value) => ++value;
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          protected override ulong Decrement(ulong value) => --value;
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected override ulong Add(ulong first, ulong second) => first + second;
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected override ulong Subtract(ulong first, ulong second) => first - second;
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
68          {
69              ref var firstLink = ref Links[first];
70              ref var secondLink = ref Links[second];
71              return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
72          }
73
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
76          {
77              ref var firstLink = ref Links[first];
78              ref var secondLink = ref Links[second];
79              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
80          }
81
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          protected override ulong GetSizeValue(ulong value) => unchecked((value & 4294967264UL)
            ↪  >> 5);
84
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
            ↪  unchecked(storedValue & 31UL | (size & 134217727UL) << 5);
87
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          protected override bool GetLeftIsChildValue(ulong value) => unchecked((value & 16UL) >>
            ↪  4 == 1UL);
90
91          [MethodImpl(MethodImplOptions.AggressiveInlining)]
92          protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
            ↪  storedValue = unchecked(storedValue & 4294967279UL | (As<bool, byte>(ref value) &
            ↪  1UL) << 4);
93
94          [MethodImpl(MethodImplOptions.AggressiveInlining)]
95          protected override bool GetRightIsChildValue(ulong value) => unchecked((value & 8UL) >>
            ↪  3 == 1UL);
96
97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
98          protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
            ↪  storedValue = unchecked(storedValue & 4294967287UL | (As<bool, byte>(ref value) &
            ↪  1UL) << 3);
99
100         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
        ↪  0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
        ↪  sbyte

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
        ↪  storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
        ↪  value & 3) & 7UL);

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

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

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

```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.ResizableDirectMemory.Generic;

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

namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
{
    public unsafe abstract class UInt64LinksSizeBalancedTreeMethodsBase :
    ↪  LinksSizeBalancedTreeMethodsBase<ulong>
    {
        protected new readonly RawLink<ulong>* Links;
        protected new readonly LinksHeader<ulong>* Header;

        protected UInt64LinksSizeBalancedTreeMethodsBase(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

```
37
38          //[MethodImpl(MethodImplOptions.AggressiveInlining)]
39          //protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override void SetLeftIsChild(ulong node, bool value) =>
            ↪  SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          protected override bool GetRightIsChild(ulong node) =>
            ↪  GetRightIsChildValue(Links[node].SizeAsSource);
46
47          //[MethodImpl(MethodImplOptions.AggressiveInlining)]
48          //protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          protected override void SetRightIsChild(ulong node, bool value) =>
            ↪  SetRightIsChildValue(ref Links[node].SizeAsSource, value);
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override sbyte GetBalance(ulong node) =>
            ↪  GetBalanceValue(Links[node].SizeAsSource);
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
            ↪  Links[node].SizeAsSource, value);
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          protected override ulong GetTreeRoot() => Header->FirstAsSource;
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget)
67              => firstSource < secondSource || (firstSource == secondSource && firstTarget <
                ↪  secondTarget);
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget)
71              => firstSource > secondSource || (firstSource == secondSource && firstTarget >
                ↪  secondTarget);
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override void ClearNode(ulong node)
75          {
76              ref var link = ref Links[node];
77              link.LeftAsSource = 0UL;
78              link.RightAsSource = 0UL;
79              link.SizeAsSource = 0UL;
80          }
81      }
82  }
```

1.48 ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
6   {
7       public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
        ↪  UInt64LinksSizeBalancedTreeMethodsBase
8       {
9           public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
            ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
            ↪  { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override ref ulong GetLeftReference(ulong node) => ref
            ↪  Links[node].LeftAsSource;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override ref ulong GetRightReference(ulong node) => ref
            ↪  Links[node].RightAsSource;
16
```

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
24          protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
            ↪ left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
            ↪ right;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsTarget);
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
            ↪ Links[node].SizeAsTarget, size);
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override bool GetLeftIsChild(ulong node) =>
            ↪ GetLeftIsChildValue(Links[node].SizeAsTarget);
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          protected override void SetLeftIsChild(ulong node, bool value) =>
            ↪ SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override bool GetRightIsChild(ulong node) =>
            ↪ GetRightIsChildValue(Links[node].SizeAsTarget);
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          protected override void SetRightIsChild(ulong node, bool value) =>
            ↪ SetRightIsChildValue(ref Links[node].SizeAsTarget, value);
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          protected override sbyte GetBalance(ulong node) =>
            ↪ GetBalanceValue(Links[node].SizeAsTarget);
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
            ↪ Links[node].SizeAsTarget, value);
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override ulong GetTreeRoot() => Header->FirstAsTarget;
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
            ↪ ulong secondSource, ulong secondTarget)
61              => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
                ↪ secondSource);
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
            ↪ ulong secondSource, ulong secondTarget)
65              => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
                ↪ secondSource);
66
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          protected override void ClearNode(ulong node)
69          {
70              ref var link = ref Links[node];
71              link.LeftAsTarget = 0UL;
72              link.RightAsTarget = 0UL;
73              link.SizeAsTarget = 0UL;
74          }
75      }
76  }
```

1.50 ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
6   {
7       public unsafe class UInt64LinksTargetsSizeBalancedTreeMethods :
        ↪ UInt64LinksSizeBalancedTreeMethodsBase
8       {
```

```
9           public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
            ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
            ↪  { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override ref ulong GetLeftReference(ulong node) => ref
            ↪  Links[node].LeftAsTarget;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override ref ulong GetRightReference(ulong node) => ref
            ↪  Links[node].RightAsTarget;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
            ↪  left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
            ↪  right;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
            ↪  size;
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override ulong GetTreeRoot() => Header->FirstAsTarget;
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget)
43              => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
                ↪  secondSource);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget)
47              => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
                ↪  secondSource);
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          protected override void ClearNode(ulong node)
51          {
52              ref var link = ref Links[node];
53              link.LeftAsTarget = 0UL;
54              link.RightAsTarget = 0UL;
55              link.SizeAsTarget = 0UL;
56          }
57      }
58  }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Memory;
5   using Platform.Data.Doublets.ResizableDirectMemory.Generic;
6   using Platform.Singletons;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
11  {
12      public unsafe class UInt64ResizableDirectMemoryLinks : ResizableDirectMemoryLinksBase<ulong>
13      {
14          private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
15          private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
16          private LinksHeader<ulong>* _header;
```

```
17          private RawLink<ulong>* _links;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public UInt64ResizableDirectMemoryLinks(string address) : this(address,
            ↪  DefaultLinksSizeStep) { }
21
22          /// <summary>
23          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪  минимальным шагом расширения базы данных.
24          /// </summary>
25          /// <param name="address">Полный пусть к файлу базы данных.</param>
26          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪  байтах.</param>
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
            ↪  this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪  memoryReservationStep) { }
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪  DefaultLinksSizeStep) { }
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep) : this(memory, memoryReservationStep,
            ↪  Default<LinksConstants<ulong>>.Instance, true) { }
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep, LinksConstants<ulong> constants, bool useAvlBasedIndex) :
            ↪  base(memory, memoryReservationStep, constants)
38          {
39              if (useAvlBasedIndex)
40              {
41                  _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
42                  _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
43              }
44              else
45              {
46                  _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
47                  _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
48              }
49              Init(memory, memoryReservationStep);
50          }
51
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
53          protected override void SetPointers(IResizableDirectMemory memory)
54          {
55              _header = (LinksHeader<ulong>*)memory.Pointer;
56              _links = (RawLink<ulong>*)memory.Pointer;
57              SourcesTreeMethods = _createSourceTreeMethods();
58              TargetsTreeMethods = _createTargetTreeMethods();
59              UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
60          }
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          protected override void ResetPointers()
64          {
65              base.ResetPointers();
66              _links = null;
67              _header = null;
68          }
69
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
            ↪  _links[linkIndex];
75
76          [MethodImpl(MethodImplOptions.AggressiveInlining)]
77          protected override bool AreEqual(ulong first, ulong second) => first == second;
78
79          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override bool LessThan(ulong first, ulong second) => first < second;

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.ResizableDirectMemory.Generic;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnusedLinksListMethods(RawLink<ulong>* links, LinksHeader<ulong>* header)
            : base((byte*)links, (byte*)header)
        {
            _links = links;
            _header = header;
        }

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

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

1.53 ./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];
            }
        }
    }
}
```

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

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

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

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

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

        private struct HalfDoublet
```

```
        {
            public TLink Element;
            public LinkFrequency<TLink> DoubletData;

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

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

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

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

        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
        ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
        ↪  minFrequencyToCompress, bool doInitialFrequenciesIncrement)
            : base(links)
        {
            _baseConverter = baseConverter;
            _doubletFrequenciesCache = doubletFrequenciesCache;
            if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
            {
                minFrequencyToCompress = Integer<TLink>.One;
            }
            _minFrequencyToCompress = minFrequencyToCompress;
            _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
            ResetMaxDoublet();
        }

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

        /// <remarks>
        /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
        /// Faster version (doublets' frequencies dictionary is not recreated).
        /// </remarks>
        private IList<TLink> Compress(IList<TLink> sequence)
        {
            if (sequence.IsNullOrEmpty())
            {
                return null;
            }
            if (sequence.Count == 1)
            {
                return sequence;
            }
            if (sequence.Count == 2)
            {
                return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
            }
            // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
            var copy = new HalfDoublet[sequence.Count];
            Doublet<TLink> doublet = default;
            for (var i = 1; i < sequence.Count; i++)
            {
                doublet.Source = sequence[i - 1];
                doublet.Target = sequence[i];
                LinkFrequency<TLink> data;
                if (_doInitialFrequenciesIncrement)
                {
                    data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
                }
                else
                {
                    data = _doubletFrequenciesCache.GetFrequency(ref doublet);
```

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

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

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

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

#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;

        protected LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;

        public abstract TLink Convert(IList<TLink> source);
    }
}
```

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

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

#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;
    }
}
```

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

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

#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));
    }
}
```

1.58 ./Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs

```
using Platform.Interfaces;

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

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

```
    }
}
```

1.59 ./Platform.Data.Doublets/Sequences/CriterionMatchers/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.CriterionMatchers
{
    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);
    }
}
```

1.60 ./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))
```

```
44              {
45                  right = Links.GetOrCreate(left, right);
46                  left = cursor;
47              }
48              return Links.GetOrCreate(left, right);
49          }
50      }
51  }
```

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

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

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

```
1   using System;
2   using System.Linq;
3   using System.Collections.Generic;
4   using Platform.Interfaces;
5   using Platform.Collections;
6   using Platform.Collections.Lists;
7   using Platform.Collections.Segments;
8   using Platform.Collections.Segments.Walkers;
9   using Platform.Singletons;
10  using Platform.Numbers;
11  using Platform.Data.Doublets.Unicode;
12
13  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15  namespace Platform.Data.Doublets.Sequences
16  {
17      public class DuplicateSegmentsProvider<TLink> :
        ↪  DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
        ↪  IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
18      {
19          private readonly ILinks<TLink> _links;
20          private readonly ILinks<TLink> _sequences;
21          private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
22          private BitString _visited;
23
24          private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
            ↪  IList<TLink>>>
25          {
26              private readonly IListEqualityComparer<TLink> _listComparer;
27              public ItemEquilityComparer() => _listComparer =
                ↪  Default<IListEqualityComparer<TLink>>.Instance;
28              public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
                ↪  KeyValuePair<IList<TLink>, IList<TLink>> right) =>
                ↪  _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
                ↪  right.Value);
29              public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
                ↪  (_listComparer.GetHashCode(pair.Key),
                ↪  _listComparer.GetHashCode(pair.Value)).GetHashCode();
30          }
31
32          private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
33          {
34              private readonly IListComparer<TLink> _listComparer;
35
36              public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
37
38              public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
                ↪  KeyValuePair<IList<TLink>, IList<TLink>> right)
39              {
40                  var intermediateResult = _listComparer.Compare(left.Key, right.Key);
```

```
                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.AddSkipFirstAndReturnConstant, 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.ShiftRight();
            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();
        }
    }
}
```

1.63 ./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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
        {
            var doublet = new Doublet<TLink>(source, target);
            return GetFrequency(ref doublet);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
        {
            _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
            return data;
        }

        public void IncrementFrequencies(IList<TLink> sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                IncrementFrequency(sequence[i - 1], sequence[i]);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
        {
            var doublet = new Doublet<TLink>(source, target);
            return IncrementFrequency(ref doublet);
        }

        public void PrintFrequencies(IList<TLink> sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                PrintFrequency(sequence[i - 1], sequence[i]);
            }
        }

        public void PrintFrequency(TLink source, TLink target)
        {
            var number = GetFrequency(source, target).Frequency;
            Console.WriteLine("({0},{1}) - {2}", source, target, number);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
        {
            if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
            {
                data.IncrementFrequency();
            }
            else
            {
                var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
                data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
                if (!_equalityComparer.Equals(link, default))
                {
                    data.Frequency = Arithmetic.Add(data.Frequency,
                    ↪  _frequencyCounter.Count(link));
                }
                _doubletsCache.Add(doublet, data);
            }
            return data;
        }

        public void ValidateFrequencies()
        {
            foreach (var entry in _doubletsCache)
            {
                var value = entry.Value;
                var linkIndex = value.Link;
                if (!_equalityComparer.Equals(linkIndex, default))
                {
                    var frequency = value.Frequency;
                    var count = _frequencyCounter.Count(linkIndex);
```

```
                    // TODO: Why `frequency` always greater than `count` by 1?
                    if (((_comparer.Compare(frequency, count) > 0) &&
                     ↪  (_comparer.Compare(Arithmetic.Subtract(frequency, count),
                     ↪  Integer<TLink>.One) > 0))
                     || ((_comparer.Compare(count, frequency) > 0) &&
                      ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency),
                      ↪  Integer<TLink>.One) > 0)))
                    {
                        throw new InvalidOperationException("Frequencies validation failed.");
                    }
                }
                //else
                //{
                //    if (value.Frequency > 0)
                //    {
                //        var frequency = value.Frequency;
                //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
                //        var count = _countLinkFrequency(linkIndex);

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

1.64 ./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}";
    }
}
```

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

```
using Platform.Converters;

#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;
    }
}
```

1.66 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounte

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

1.67 ./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;
```

```
        }
    }
}
```

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

```
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();
    }
}
```

1.69 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffC

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

1.70 ./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();
    }
}
```

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

```
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;
        }
    }
}
```

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

```

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

#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 IProperties<TLink, TLink, TLink> _propertyOperator;

        public CachedSequenceHeightProvider(
            ILinks<TLink> links,
            ISequenceHeightProvider<TLink> baseHeightProvider,
            IConverter<TLink> addressToUnaryNumberConverter,
            IConverter<TLink> unaryNumberToAddressConverter,
            TLink heightPropertyMarker,
            IProperties<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;
        }
    }
}
```

1.73 ./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;
        }
    }
}
```

1.74 ./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>
    {
```

```
    }
}
```

1.75 ./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);
        }
    }
}
```

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

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

#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 IProperty<TLink, TLink> _frequencyPropertyOperator;
        private readonly IIncrementer<TLink> _frequencyIncrementer;

        public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IProperty<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);
        }
    }
}
```

1.77 ./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);
    }
}
```

1.78 ./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;
        }
    }
}
```

1.79 ./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;
            });
        }
    }
}
```

1.80 ./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;
    }
}
```

1.81 ./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.Collections.Sets;
using Platform.Collections.Stacks;
using Platform.Data.Exceptions;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Walkers;

#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 Array.Empty<ulong>();
                }
                Links.EnsureLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return sequence;
                }
                return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
            });
        }

        private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
        {
#if DEBUG
            if ((stopAt - startAt) < 0)
            {
                throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
                ↪  меньше или равен stopAt");
            }
#endif
            if ((stopAt - startAt) == 0)
            {
                return new[] { sequence[startAt] };
            }
            if ((stopAt - startAt) == 1)
            {
```

```
                return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
            }
            var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
            var last = 0;
            for (var splitter = startAt; splitter < stopAt; splitter++)
            {
                var left = CreateAllVariants2Core(sequence, startAt, splitter);
                var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
                for (var i = 0; i < left.Length; i++)
                {
                    for (var j = 0; j < right.Length; j++)
                    {
                        var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
                        if (variant == Constants.Null)
                        {
                            throw new NotImplementedException("Creation cancellation is not
                            ↪  implemented.");
                        }
                        variants[last++] = variant;
                    }
                }
            }
            return variants;
        }

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

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

```
            return results;
        }

        #endregion

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

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

        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.EnsureLinkIsAnyOrExists(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;
            });
        }
```

```
288
289         private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
290         {
291             Links.Unsync.Each(left, Constants.Any, rightStep =>
292             {
293                 TryStepRightUp(handler, right, rightStep);
294                 return true;
295             });
296         }
297
298         private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
            ↪ stepFrom)
299         {
300             var upStep = stepFrom;
301             var firstSource = Links.Unsync.GetTarget(upStep);
302             while (firstSource != right && firstSource != upStep)
303             {
304                 upStep = firstSource;
305                 firstSource = Links.Unsync.GetSource(upStep);
306             }
307             if (firstSource == right)
308             {
309                 handler(new LinkAddress<LinkIndex>(stepFrom));
310             }
311         }
312
313         // TODO: Test
314         private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
315         {
316             Links.Unsync.Each(right, Constants.Any, doublet =>
317             {
318                 StepLeft(handler, left, doublet);
319                 if (right != doublet)
320                 {
321                     PartialStepLeft(handler, left, doublet);
322                 }
323                 return true;
324             });
325         }
326
327         private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
328         {
329             Links.Unsync.Each(Constants.Any, right, leftStep =>
330             {
331                 TryStepLeftUp(handler, left, leftStep);
332                 return true;
333             });
334         }
335
336         private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
337         {
338             var upStep = stepFrom;
339             var firstTarget = Links.Unsync.GetSource(upStep);
340             while (firstTarget != left && firstTarget != upStep)
341             {
342                 upStep = firstTarget;
343                 firstTarget = Links.Unsync.GetTarget(upStep);
344             }
345             if (firstTarget == left)
346             {
347                 handler(new LinkAddress<LinkIndex>(stepFrom));
348             }
349         }
350
351         private bool StartsWith(ulong sequence, ulong link)
352         {
353             var upStep = sequence;
354             var firstSource = Links.Unsync.GetSource(upStep);
355             while (firstSource != link && firstSource != upStep)
356             {
357                 upStep = firstSource;
358                 firstSource = Links.Unsync.GetSource(upStep);
359             }
360             return firstSource == link;
361         }
362
363         private bool EndsWith(ulong sequence, ulong link)
364         {
365             var upStep = sequence;
```

```
366             var lastTarget = Links.Unsync.GetTarget(upStep);
367             while (lastTarget != link && lastTarget != upStep)
368             {
369                 upStep = lastTarget;
370                 lastTarget = Links.Unsync.GetTarget(upStep);
371             }
372             return lastTarget == link;
373         }
374 
375         public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
376         {
377             return _sync.ExecuteReadOperation(() =>
378             {
379                 var results = new List<ulong>();
380                 if (sequence.Length > 0)
381                 {
382                     Links.EnsureLinkExists(sequence);
383                     var firstElement = sequence[0];
384                     if (sequence.Length == 1)
385                     {
386                         results.Add(firstElement);
387                         return results;
388                     }
389                     if (sequence.Length == 2)
390                     {
391                         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
392                         if (doublet != Constants.Null)
393                         {
394                             results.Add(doublet);
395                         }
396                         return results;
397                     }
398                     var linksInSequence = new HashSet<ulong>(sequence);
399                     void handler(IList<LinkIndex> result)
400                     {
401                         var resultIndex = result[Links.Constants.IndexPart];
402                         var filterPosition = 0;
403                         StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
                            ↪  Links.Unsync.GetTarget,
404                             x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                                ↪  x =>
405                             {
406                                 if (filterPosition == sequence.Length)
407                                 {
408                                     filterPosition = -2; // Длиннее чем нужно
409                                     return false;
410                                 }
411                                 if (x != sequence[filterPosition])
412                                 {
413                                     filterPosition = -1;
414                                     return false; // Начинается иначе
415                                 }
416                                 filterPosition++;
417 
418                                 return true;
419                             });
420                         if (filterPosition == sequence.Length)
421                         {
422                             results.Add(resultIndex);
423                         }
424                     }
425                     if (sequence.Length >= 2)
426                     {
427                         StepRight(handler, sequence[0], sequence[1]);
428                     }
429                     var last = sequence.Length - 2;
430                     for (var i = 1; i < last; i++)
431                     {
432                         PartialStepRight(handler, sequence[i], sequence[i + 1]);
433                     }
434                     if (sequence.Length >= 3)
435                     {
436                         StepLeft(handler, sequence[sequence.Length - 2],
                            ↪  sequence[sequence.Length - 1]);
437                     }
438                 }
439                 return results;
440             });
441         }
```

```
442
443         public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
444         {
445             return _sync.ExecuteReadOperation(() =>
446             {
447                 var results = new HashSet<ulong>();
448                 if (sequence.Length > 0)
449                 {
450                     Links.EnsureLinkExists(sequence);
451                     var firstElement = sequence[0];
452                     if (sequence.Length == 1)
453                     {
454                         results.Add(firstElement);
455                         return results;
456                     }
457                     if (sequence.Length == 2)
458                     {
459                         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
460                         if (doublet != Constants.Null)
461                         {
462                             results.Add(doublet);
463                         }
464                         return results;
465                     }
466                     var matcher = new Matcher(this, sequence, results, null);
467                     if (sequence.Length >= 2)
468                     {
469                         StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
470                     }
471                     var last = sequence.Length - 2;
472                     for (var i = 1; i < last; i++)
473                     {
474                         PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
                             ↪ sequence[i + 1]);
475                     }
476                     if (sequence.Length >= 3)
477                     {
478                         StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
                             ↪ sequence[sequence.Length - 1]);
479                     }
480                 }
481                 return results;
482             });
483         }
484
485         public const int MaxSequenceFormatSize = 200;
486
487         public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
            ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
488
489         public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
            ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
            ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
            ↪ elementToString, insertComma, knownElements));
490
491         private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
            ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
            ↪ LinkIndex[] knownElements)
492         {
493             var linksInSequence = new HashSet<ulong>(knownElements);
494             //var entered = new HashSet<ulong>();
495             var sb = new StringBuilder();
496             sb.Append('{');
497             if (links.Exists(sequenceLink))
498             {
499                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
500                     x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
                         ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
501                     {
502                         if (insertComma && sb.Length > 1)
503                         {
504                             sb.Append(',');
505                         }
506                         //if (entered.Contains(element))
507                         //{
508                         //    sb.Append('{');
509                         //    elementToString(sb, element);
510                         //    sb.Append('}');
```

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

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

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

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

    public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            if (sequence.Length > 0)
            {
                Links.EnsureLinkExists(sequence);
                var results = new HashSet<ulong>();
                for (var i = 0; i < sequence.Length; i++)
                {
                    AllUsagesCore(sequence[i], results);
                }
                var filteredResults = new List<ulong>();
                var linksInSequence = new HashSet<ulong>(sequence);
```

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

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

```
                    //{
                    //    AllUsagesCore(sequence[i], nextResults);
                    //    if (results.IsNullOrEmpty())
                    //    {
                    //        results = nextResults;
                    //        nextResults = new HashSet<ulong>();
                    //    }
                    //    else
                    //    {
                    //        results.IntersectWith(nextResults);
                    //        nextResults.Clear();
                    //    }
                    //}
                    var collector1 = new AllUsagesCollector1(Links.Unsync, results);
                    collector1.Collect(Links.Unsync.GetLink(sequence[0]));
                    var next = new HashSet<ulong>();
                    for (var i = 1; i < sequence.Count; i++)
                    {
                        var collector = new AllUsagesCollector1(Links.Unsync, next);
                        collector.Collect(Links.Unsync.GetLink(sequence[i]));

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

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

        public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    //var firstElement = sequence[0];
                    //if (sequence.Length == 1)
                    //{
                    //    //results.Add(firstElement);
                    //    return results;
                    //}
                    //if (sequence.Length == 2)
                    //{
                    //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
                    //    //if (doublet != Doublets.Links.Null)
                    //    //    results.Add(doublet);
                    //    return results;
                    //}
                    //var lastElement = sequence[sequence.Length - 1];
                    //Func<ulong, bool> handler = x =>
                    //{
                    //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
                    ↪  results.Add(x);
                    //    return true;
```

```
                    //};
                    //if (sequence.Length >= 2)
                    //    StepRight(handler, sequence[0], sequence[1]);
                    //var last = sequence.Length - 2;
                    //for (var i = 1; i < last; i++)
                    //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
                    //if (sequence.Length >= 3)
                    //    StepLeft(handler, sequence[sequence.Length - 2],
                    ↪  sequence[sequence.Length - 1]);
                    //////if (sequence.Length == 1)
                    //////{
                    //////    throw new NotImplementedException(); // all sequences, containing
                    ↪  this element?
                    //////}
                    //////if (sequence.Length == 2)
                    //////{
                    //////    var results = new List<ulong>();
                    //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
                    //////    return results;
                    //////}
                    //////var matches = new List<List<ulong>>();
                    //////var last = sequence.Length - 1;
                    //////for (var i = 0; i < last; i++)
                    //////{
                    //////    var results = new List<ulong>();
                    //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
                    //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
                    //////    if (results.Count > 0)
                    //////        matches.Add(results);
                    //////    else
                    //////        return results;
                    //////    if (matches.Count == 2)
                    //////    {
                    //////        var merged = new List<ulong>();
                    //////        for (var j = 0; j < matches[0].Count; j++)
                    //////            for (var k = 0; k < matches[1].Count; k++)
                    //////                CloseInnerConnections(merged.Add, matches[0][j],
                    ↪  matches[1][k]);
                    //////        if (merged.Count > 0)
                    //////            matches = new List<List<ulong>> { merged };
                    //////        else
                    //////            return new List<ulong>();
                    //////    }
                    //////}
                    //////if (matches.Count > 0)
                    //////{
                    //////    var usages = new HashSet<ulong>();
                    //////    for (int i = 0; i < sequence.Length; i++)
                    //////    {
                    //////        AllUsagesCore(sequence[i], usages);
                    //////    }
                    //////    //for (int i = 0; i < matches[0].Count; i++)
                    //////    //    AllUsagesCore(matches[0][i], usages);
                    //////    //usages.UnionWith(matches[0]);
                    //////    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
```

```
877             /// </remarks>
878             public HashSet<ulong> AllUsages(ulong link)
879             {
880                 return _sync.ExecuteReadOperation(() =>
881                 {
882                     var usages = new HashSet<ulong>();
883                     AllUsagesCore(link, usages);
884                     return usages;
885                 });
886             }
887 
888             // При сборе всех использований (последовательностей) можно сохранять обратный путь к
                ↪  той связи с которой начинался поиск (STTTSSSTT),
889             // причём достаточно одного бита для хранения перехода влево или вправо
890             private void AllUsagesCore(ulong link, HashSet<ulong> usages)
891             {
892                 bool handler(ulong doublet)
893                 {
894                     if (usages.Add(doublet))
895                     {
896                         AllUsagesCore(doublet, usages);
897                     }
898                     return true;
899                 }
900                 Links.Unsync.Each(link, Constants.Any, handler);
901                 Links.Unsync.Each(Constants.Any, link, handler);
902             }
903 
904             public HashSet<ulong> AllBottomUsages(ulong link)
905             {
906                 return _sync.ExecuteReadOperation(() =>
907                 {
908                     var visits = new HashSet<ulong>();
909                     var usages = new HashSet<ulong>();
910                     AllBottomUsagesCore(link, visits, usages);
911                     return usages;
912                 });
913             }
914 
915             private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
                ↪  usages)
916             {
917                 bool handler(ulong doublet)
918                 {
919                     if (visits.Add(doublet))
920                     {
921                         AllBottomUsagesCore(doublet, visits, usages);
922                     }
923                     return true;
924                 }
925                 if (Links.Unsync.Count(Constants.Any, link) == 0)
926                 {
927                     usages.Add(link);
928                 }
929                 else
930                 {
931                     Links.Unsync.Each(link, Constants.Any, handler);
932                     Links.Unsync.Each(Constants.Any, link, handler);
933                 }
934             }
935 
936             public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
937             {
938                 if (Options.UseSequenceMarker)
939                 {
940                     var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                        ↪  Options.MarkedSequenceMatcher, symbol);
941                     return counter.Count();
942                 }
943                 else
944                 {
945                     var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                        ↪  symbol);
946                     return counter.Count();
947                 }
948             }
949 
```

```
950         private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
            ↪ LinkIndex> outerHandler)
951         {
952             bool handler(ulong doublet)
953             {
954                 if (usages.Add(doublet))
955                 {
956                     if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
957                     {
958                         return false;
959                     }
960                     if (!AllUsagesCore1(doublet, usages, outerHandler))
961                     {
962                         return false;
963                     }
964                 }
965                 return true;
966             }
967             return Links.Unsync.Each(link, Constants.Any, handler)
968                 && Links.Unsync.Each(Constants.Any, link, handler);
969         }
970 
971         public void CalculateAllUsages(ulong[] totals)
972         {
973             var calculator = new AllUsagesCalculator(Links, totals);
974             calculator.Calculate();
975         }
976 
977         public void CalculateAllUsages2(ulong[] totals)
978         {
979             var calculator = new AllUsagesCalculator2(Links, totals);
980             calculator.Calculate();
981         }
982 
983         private class AllUsagesCalculator
984         {
985             private readonly SynchronizedLinks<ulong> _links;
986             private readonly ulong[] _totals;
987 
988             public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
989             {
990                 _links = links;
991                 _totals = totals;
992             }
993 
994             public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
                ↪ CalculateCore);
995 
996             private bool CalculateCore(ulong link)
997             {
998                 if (_totals[link] == 0)
999                 {
1000                     var total = 1UL;
1001                     _totals[link] = total;
1002                     var visitedChildren = new HashSet<ulong>();
1003                     bool linkCalculator(ulong child)
1004                     {
1005                         if (link != child && visitedChildren.Add(child))
1006                         {
1007                             total += _totals[child] == 0 ? 1 : _totals[child];
1008                         }
1009                         return true;
1010                     }
1011                     _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1012                     _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1013                     _totals[link] = total;
1014                 }
1015                 return true;
1016             }
1017         }
1018 
1019         private class AllUsagesCalculator2
1020         {
1021             private readonly SynchronizedLinks<ulong> _links;
1022             private readonly ulong[] _totals;
1023 
1024             public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1025             {
1026                 _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
    {
```

```
1104            private readonly ILinks<ulong> _links;
1105            private readonly HashSet<ulong> _usages;
1106
1107            public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1108            {
1109                _links = links;
1110                _usages = usages;
1111            }
1112
1113            public bool Collect(ulong link)
1114            {
1115                if (_usages.Add(link))
1116                {
1117                    _links.Each(link, _links.Constants.Any, Collect);
1118                    _links.Each(_links.Constants.Any, link, Collect);
1119                }
1120                return true;
1121            }
1122        }
1123
1124        private class AllUsagesCollector1
1125        {
1126            private readonly ILinks<ulong> _links;
1127            private readonly HashSet<ulong> _usages;
1128            private readonly ulong _continue;
1129
1130            public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1131            {
1132                _links = links;
1133                _usages = usages;
1134                _continue = _links.Constants.Continue;
1135            }
1136
1137            public ulong Collect(IList<ulong> link)
1138            {
1139                var linkIndex = _links.GetIndex(link);
1140                if (_usages.Add(linkIndex))
1141                {
1142                    _links.Each(Collect, _links.Constants.Any, linkIndex);
1143                }
1144                return _continue;
1145            }
1146        }
1147
1148        private class AllUsagesCollector2
1149        {
1150            private readonly ILinks<ulong> _links;
1151            private readonly BitString _usages;
1152
1153            public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1154            {
1155                _links = links;
1156                _usages = usages;
1157            }
1158
1159            public bool Collect(ulong link)
1160            {
1161                if (_usages.Add((long)link))
1162                {
1163                    _links.Each(link, _links.Constants.Any, Collect);
1164                    _links.Each(_links.Constants.Any, link, Collect);
1165                }
1166                return true;
1167            }
1168        }
1169
1170        private class AllUsagesIntersectingCollector
1171        {
1172            private readonly SynchronizedLinks<ulong> _links;
1173            private readonly HashSet<ulong> _intersectWith;
1174            private readonly HashSet<ulong> _usages;
1175            private readonly HashSet<ulong> _enter;
1176
1177            public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
                ↪ intersectWith, HashSet<ulong> usages)
1178            {
1179                _links = links;
1180                _intersectWith = intersectWith;
1181                _usages = usages;
1182                _enter = new HashSet<ulong>(); // защита от зацикливания
1183            }
```

```
1184
1185                public bool Collect(ulong link)
1186                {
1187                    if (_enter.Add(link))
1188                    {
1189                        if (_intersectWith.Contains(link))
1190                        {
1191                            _usages.Add(link);
1192                        }
1193                        _links.Unsync.Each(link, _links.Constants.Any, Collect);
1194                        _links.Unsync.Each(_links.Constants.Any, link, Collect);
1195                    }
1196                    return true;
1197                }
1198            }
1199
1200            private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
                ↪ right)
1201            {
1202                TryStepLeftUp(handler, left, right);
1203                TryStepRightUp(handler, right, left);
1204            }
1205
1206            private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
                ↪ right)
1207            {
1208                // Direct
1209                if (left == right)
1210                {
1211                    handler(new LinkAddress<LinkIndex>(left));
1212                }
1213                var doublet = Links.Unsync.SearchOrDefault(left, right);
1214                if (doublet != Constants.Null)
1215                {
1216                    handler(new LinkAddress<LinkIndex>(doublet));
1217                }
1218                // Inner
1219                CloseInnerConnections(handler, left, right);
1220                // Outer
1221                StepLeft(handler, left, right);
1222                StepRight(handler, left, right);
1223                PartialStepRight(handler, left, right);
1224                PartialStepLeft(handler, left, right);
1225            }
1226
1227            private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
                ↪ HashSet<ulong> previousMatchings, long startAt)
1228            {
1229                if (startAt >= sequence.Length) // ?
1230                {
1231                    return previousMatchings;
1232                }
1233                var secondLinkUsages = new HashSet<ulong>();
1234                AllUsagesCore(sequence[startAt], secondLinkUsages);
1235                secondLinkUsages.Add(sequence[startAt]);
1236                var matchings = new HashSet<ulong>();
1237                var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1238                //for (var i = 0; i < previousMatchings.Count; i++)
1239                foreach (var secondLinkUsage in secondLinkUsages)
1240                {
1241                    foreach (var previousMatching in previousMatchings)
1242                    {
1243                        //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
                            ↪ secondLinkUsage);
1244                        StepRight(filler.AddFirstAndReturnConstant, previousMatching,
                            ↪ secondLinkUsage);
1245                        TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
                            ↪ previousMatching);
1246                        //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
                            ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
                            ↪ желаемым результам.
1247                        PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
                            ↪ secondLinkUsage);
1248                    }
1249                }
1250                if (matchings.Count == 0)
1251                {
```

```
1252                return matchings;
1253            }
1254            return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1255        }
1256
1257        private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
        ↪  links, params ulong[] sequence)
1258        {
1259            if (sequence == null)
1260            {
1261                return;
1262            }
1263            for (var i = 0; i < sequence.Length; i++)
1264            {
1265                if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
                    ↪  !links.Exists(sequence[i]))
1266                {
1267                    throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                        ↪  $"patternSequence[{i}]");
1268                }
1269            }
1270        }
1271
1272        // Pattern Matching -> Key To Triggers
1273        public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1274        {
1275            return _sync.ExecuteReadOperation(() =>
1276            {
1277                patternSequence = Simplify(patternSequence);
1278                if (patternSequence.Length > 0)
1279                {
1280                    EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1281                    var uniqueSequenceElements = new HashSet<ulong>();
1282                    for (var i = 0; i < patternSequence.Length; i++)
1283                    {
1284                        if (patternSequence[i] != Constants.Any && patternSequence[i] !=
                            ↪  ZeroOrMany)
1285                        {
1286                            uniqueSequenceElements.Add(patternSequence[i]);
1287                        }
1288                    }
1289                    var results = new HashSet<ulong>();
1290                    foreach (var uniqueSequenceElement in uniqueSequenceElements)
1291                    {
1292                        AllUsagesCore(uniqueSequenceElement, results);
1293                    }
1294                    var filteredResults = new HashSet<ulong>();
1295                    var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1296                    matcher.AddAllPatternMatchedToResults(results);
1297                    return filteredResults;
1298                }
1299                return new HashSet<ulong>();
1300            });
1301        }
1302
1303        // Найти все возможные связи между указанным списком связей.
1304        // Находит связи между всеми указанными связями в любом порядке.
1305        // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
        ↪  несколько раз в последовательности)
1306        public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1307        {
1308            return _sync.ExecuteReadOperation(() =>
1309            {
1310                var results = new HashSet<ulong>();
1311                if (linksToConnect.Length > 0)
1312                {
1313                    Links.EnsureLinkExists(linksToConnect);
1314                    AllUsagesCore(linksToConnect[0], results);
1315                    for (var i = 1; i < linksToConnect.Length; i++)
1316                    {
1317                        var next = new HashSet<ulong>();
1318                        AllUsagesCore(linksToConnect[i], next);
1319                        results.IntersectWith(next);
1320                    }
1321                }
1322                return results;
1323            });
1324        }
```

```
1325
1326        public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1327        {
1328            return _sync.ExecuteReadOperation(() =>
1329            {
1330                var results = new HashSet<ulong>();
1331                if (linksToConnect.Length > 0)
1332                {
1333                    Links.EnsureLinkExists(linksToConnect);
1334                    var collector1 = new AllUsagesCollector(Links.Unsync, results);
1335                    collector1.Collect(linksToConnect[0]);
1336                    var next = new HashSet<ulong>();
1337                    for (var i = 1; i < linksToConnect.Length; i++)
1338                    {
1339                        var collector = new AllUsagesCollector(Links.Unsync, next);
1340                        collector.Collect(linksToConnect[i]);
1341                        results.IntersectWith(next);
1342                        next.Clear();
1343                    }
1344                }
1345                return results;
1346            });
1347        }
1348
1349        public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1350        {
1351            return _sync.ExecuteReadOperation(() =>
1352            {
1353                var results = new HashSet<ulong>();
1354                if (linksToConnect.Length > 0)
1355                {
1356                    Links.EnsureLinkExists(linksToConnect);
1357                    var collector1 = new AllUsagesCollector(Links, results);
1358                    collector1.Collect(linksToConnect[0]);
1359                    //AllUsagesCore(linksToConnect[0], results);
1360                    for (var i = 1; i < linksToConnect.Length; i++)
1361                    {
1362                        var next = new HashSet<ulong>();
1363                        var collector = new AllUsagesIntersectingCollector(Links, results, next);
1364                        collector.Collect(linksToConnect[i]);
1365                        //AllUsagesCore(linksToConnect[i], next);
1366                        //results.IntersectWith(next);
1367                        results = next;
1368                    }
1369                }
1370                return results;
1371            });
1372        }
1373
1374        public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1375        {
1376            return _sync.ExecuteReadOperation(() =>
1377            {
1378                var results = new BitString((long)Links.Unsync.Count() + 1); // new
                    ↪  BitArray((int)_links.Total + 1);
1379                if (linksToConnect.Length > 0)
1380                {
1381                    Links.EnsureLinkExists(linksToConnect);
1382                    var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1383                    collector1.Collect(linksToConnect[0]);
1384                    for (var i = 1; i < linksToConnect.Length; i++)
1385                    {
1386                        var next = new BitString((long)Links.Unsync.Count() + 1); //new
                            ↪  BitArray((int)_links.Total + 1);
1387                        var collector = new AllUsagesCollector2(Links.Unsync, next);
1388                        collector.Collect(linksToConnect[i]);
1389                        results = results.And(next);
1390                    }
1391                }
1392                return results.GetSetUInt64Indices();
1393            });
1394        }
1395
1396        private static ulong[] Simplify(ulong[] sequence)
1397        {
1398            // Считаем новый размер последовательности
1399            long newLength = 0;
1400            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>();

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

        #region From Triplets

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

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

```
            var left = links[leftBound++];
            var right = links[rightBound--];
            var results = new List<ulong>();
            CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
            return results;
        }

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

        public ulong[] GetRightElements(ulong startLink, ulong rightLink)
        {
            var result = new ulong[5];
            TryStepRight(startLink, rightLink, result, 0);
            Links.Each(Constants.Any, startLink, couple =>
            {
                if (couple != startLink)
                {
```

```
                if (TryStepRight(couple, rightLink, result, 2))
                {
                    return false;
                }
            }
            return true;
        });
        if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
        {
            result[4] = startLink;
        }
        return result;
    }

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

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

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

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

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

```
1787                }
1788                if (patternBlock.Type != PatternBlockType.Undefined)
1789                {
1790                    pattern.Add(patternBlock);
1791                }
1792                return pattern;
1793            }
1794
1795            // match: search for regexp anywhere in text
1796            //int match(char* regexp, char* text)
1797            //{
1798            //    do
1799            //    {
1800            //    } while (*text++ != '\0');
1801            //    return 0;
1802            //}
1803
1804            // matchhere: search for regexp at beginning of text
1805            //int matchhere(char* regexp, char* text)
1806            //{
1807            //    if (regexp[0] == '\0')
1808            //        return 1;
1809            //    if (regexp[1] == '*')
1810            //        return matchstar(regexp[0], regexp + 2, text);
1811            //    if (regexp[0] == '$' && regexp[1] == '\0')
1812            //        return *text == '\0';
1813            //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1814            //        return matchhere(regexp + 1, text + 1);
1815            //    return 0;
1816            //}
1817
1818            // matchstar: search for c*regexp at beginning of text
1819            //int matchstar(int c, char* regexp, char* text)
1820            //{
1821            //    do
1822            //    {    /* a * matches zero or more instances */
1823            //        if (matchhere(regexp, text))
1824            //            return 1;
1825            //    } while (*text != '\0' && (*text++ == c || c == '.'));
1826            //    return 0;
1827            //}
1828
1829            //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
                ↪  long maximumGap)
1830            //{
1831            //    mininumGap = 0;
1832            //    maximumGap = 0;
1833            //    element = 0;
1834            //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1835            //    {
1836            //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1837            //            mininumGap++;
1838            //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1839            //            maximumGap = long.MaxValue;
1840            //        else
1841            //            break;
1842            //    }
1843
1844            //    if (maximumGap < mininumGap)
1845            //        maximumGap = mininumGap;
1846            //}
1847
1848            private bool PatternMatchCore(LinkIndex element)
1849            {
1850                if (_patternPosition >= _pattern.Count)
1851                {
1852                    _patternPosition = -2;
1853                    return false;
1854                }
1855                var currentPatternBlock = _pattern[_patternPosition];
1856                if (currentPatternBlock.Type == PatternBlockType.Gap)
1857                {
1858                    //var currentMatchingBlockLength = (_sequencePosition -
                        ↪  _lastMatchedBlockPosition);
1859                    if (_sequencePosition < currentPatternBlock.Start)
1860                    {
1861                        _sequencePosition++;
1862                        return true; // Двигаемся дальше
```

```
                    }
                    // Это последний блок
                    if (_pattern.Count == _patternPosition + 1)
                    {
                        _patternPosition++;
                        _sequencePosition = 0;
                        return false; // Полное соответствие
                    }
                    else
                    {
                        if (_sequencePosition > currentPatternBlock.Stop)
                        {
                            return false; // Соответствие невозможно
                        }
                        var nextPatternBlock = _pattern[_patternPosition + 1];
                        if (_patternSequence[nextPatternBlock.Start] == element)
                        {
                            if (nextPatternBlock.Start < nextPatternBlock.Stop)
                            {
                                _patternPosition++;
                                _sequencePosition = 1;
                            }
                            else
                            {
                                _patternPosition += 2;
                                _sequencePosition = 0;
                            }
                        }
                    }
                }
                else // currentPatternBlock.Type == PatternBlockType.Elements
                {
                    var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
                    if (_patternSequence[patternElementPosition] != element)
                    {
                        return false; // Соответствие невозможно
                    }
                    if (patternElementPosition == currentPatternBlock.Stop)
                    {
                        _patternPosition++;
                        _sequencePosition = 0;
                    }
                    else
                    {
                        _sequencePosition++;
                    }
                }
                return true;
                //if (_patternSequence[_patternPosition] != element)
                //    return false;
                //else
                //{
                //    _sequencePosition++;
                //    _patternPosition++;
                //    return true;
                //}
                ////////
                //if (_filterPosition == _patternSequence.Length)
                //{
                //    _filterPosition = -2; // Длиннее чем нужно
                //    return false;
                //}
                //if (element != _patternSequence[_filterPosition])
                //{
                //    _filterPosition = -1;
                //    return false; // Начинается иначе
                //}
                //_filterPosition++;
                //if (_filterPosition == (_patternSequence.Length - 1))
                //    return false;
                //if (_filterPosition >= 0)
                //{
                //    if (element == _patternSequence[_filterPosition + 1])
                //        _filterPosition++;
                //    else
                //        return false;
                //}
                //if (_filterPosition < 0)
                //{
```

```
                //    if (element == _patternSequence[0])
                //        _filterPosition = 0;
                //}
            }

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

        #endregion
    }
}
```

1.82 ./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.Collections.Stacks;
using Platform.Threading.Synchronization;
using Platform.Data.Doublets.Sequences.Walkers;
using LinkIndex = System.UInt64;

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

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

```
52          /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
            ↪  связей.</summary>
53          public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
54
55          public SequencesOptions<LinkIndex> Options { get; }
56          public SynchronizedLinks<LinkIndex> Links { get; }
57          private readonly ISynchronization _sync;
58
59          public LinksConstants<LinkIndex> Constants { get; }
60
61          public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
62          {
63              Links = links;
64              _sync = links.SyncRoot;
65              Options = options;
66              Options.ValidateOptions();
67              Options.InitOptions(Links);
68              Constants = links.Constants;
69          }
70
71          public Sequences(SynchronizedLinks<LinkIndex> links)
72              : this(links, new SequencesOptions<LinkIndex>())
73          {
74          }
75
76          public bool IsSequence(LinkIndex sequence)
77          {
78              return _sync.ExecuteReadOperation(() =>
79              {
80                  if (Options.UseSequenceMarker)
81                  {
82                      return Options.MarkedSequenceMatcher.IsMatched(sequence);
83                  }
84                  return !Links.Unsync.IsPartialPoint(sequence);
85              });
86          }
87
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          private LinkIndex GetSequenceByElements(LinkIndex sequence)
90          {
91              if (Options.UseSequenceMarker)
92              {
93                  return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
94              }
95              return sequence;
96          }
97
98          private LinkIndex GetSequenceElements(LinkIndex sequence)
99          {
100             if (Options.UseSequenceMarker)
101             {
102                 var linkContents = new Link<ulong>(Links.GetLink(sequence));
103                 if (linkContents.Source == Options.SequenceMarkerLink)
104                 {
105                     return linkContents.Target;
106                 }
107                 if (linkContents.Target == Options.SequenceMarkerLink)
108                 {
109                     return linkContents.Source;
110                 }
111             }
112             return sequence;
113         }
114
115         #region Count
116
117         public LinkIndex Count(IList<LinkIndex> restrictions)
118         {
119             if (restrictions.IsNullOrEmpty())
120             {
121                 return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
122             }
123             if (restrictions.Count == 1) // Первая связь это адрес
124             {
125                 var sequenceIndex = restrictions[0];
126                 if (sequenceIndex == Constants.Null)
127                 {
128                     return 0;
129                 }
```

```
            if (sequenceIndex == Constants.Any)
            {
                return Count(null);
            }
            if (Options.UseSequenceMarker)
            {
                return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
            }
            return Links.Exists(sequenceIndex) ? 1UL : 0;
        }
        throw new NotImplementedException();
    }

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

    #endregion

    #region Create

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

    private LinkIndex CreateCore(IList<LinkIndex> restrictions)
    {
        LinkIndex[] sequence = restrictions.SkipFirst();
        if (Options.UseIndex)
        {
            Options.Index.Add(sequence);
        }
        var sequenceRoot = default(LinkIndex);
        if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
        {
            var matches = Each(restrictions);
            if (matches.Count > 0)
            {
                sequenceRoot = matches[0];
            }
        }
        else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
        {
            return CompactCore(sequence);
```

```
207             }
208             if (sequenceRoot == default)
209             {
210                 sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
211             }
212             if (Options.UseSequenceMarker)
213             {
214                 return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
215             }
216             return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
217         }
218 
219         #endregion
220 
221         #region Each
222 
223         public List<LinkIndex> Each(IList<LinkIndex> sequence)
224         {
225             var results = new List<LinkIndex>();
226             var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
227             Each(filler.AddFirstAndReturnConstant, sequence);
228             return results;
229         }
230 
231         public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
         ↪  restrictions)
232         {
233             return _sync.ExecuteReadOperation(() =>
234             {
235                 if (restrictions.IsNullOrEmpty())
236                 {
237                     return Constants.Continue;
238                 }
239                 Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
240                 if (restrictions.Count == 1)
241                 {
242                     var link = restrictions[0];
243                     var any = Constants.Any;
244                     if (link == any)
245                     {
246                         if (Options.UseSequenceMarker)
247                         {
248                             return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
                             ↪  Options.SequenceMarkerLink, any));
249                         }
250                         else
251                         {
252                             return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
                             ↪  any));
253                         }
254                     }
255                     if (Options.UseSequenceMarker)
256                     {
257                         var sequenceLinkValues = Links.Unsync.GetLink(link);
258                         if (sequenceLinkValues[Constants.SourcePart] ==
                         ↪  Options.SequenceMarkerLink)
259                         {
260                             link = sequenceLinkValues[Constants.TargetPart];
261                         }
262                     }
263                     var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
264                     sequence[0] = link;
265                     return handler(sequence);
266                 }
267                 else if (restrictions.Count == 2)
268                 {
269                     throw new NotImplementedException();
270                 }
271                 else if (restrictions.Count == 3)
272                 {
273                     return Links.Unsync.Each(handler, restrictions);
274                 }
275                 else
276                 {
277                     var sequence = restrictions.SkipFirst();
278                     if (Options.UseIndex && !Options.Index.MightContain(sequence))
279                     {
280                         return Constants.Break;
```

```
281                     }
282                     return EachCore(handler, sequence);
283                 }
284             });
285         }
286 
287         private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
        ↪  values)
288         {
289             var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
290             // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
            ↪  Id.
291             Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
            ↪  (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
            ↪  matcher.HandleFullMatched;
292             //if (sequence.Length >= 2)
293             if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
294             {
295                 return Constants.Break;
296             }
297             var last = values.Count - 2;
298             for (var i = 1; i < last; i++)
299             {
300                 if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
                    ↪  Constants.Continue)
301                 {
302                     return Constants.Break;
303                 }
304             }
305             if (values.Count >= 3)
306             {
307                 if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
                    ↪  != Constants.Continue)
308                 {
309                     return Constants.Break;
310                 }
311             }
312             return Constants.Continue;
313         }
314 
315         private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
        ↪  left, LinkIndex right)
316         {
317             return Links.Unsync.Each(doublet =>
318             {
319                 var doubletIndex = doublet[Constants.IndexPart];
320                 if (StepRight(handler, doubletIndex, right) != Constants.Continue)
321                 {
322                     return Constants.Break;
323                 }
324                 if (left != doubletIndex)
325                 {
326                     return PartialStepRight(handler, doubletIndex, right);
327                 }
328                 return Constants.Continue;
329             }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
330         }
331 
332         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));
333 
334         private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
        ↪  right, LinkIndex stepFrom)
335         {
336             var upStep = stepFrom;
337             var firstSource = Links.Unsync.GetTarget(upStep);
338             while (firstSource != right && firstSource != upStep)
339             {
340                 upStep = firstSource;
341                 firstSource = Links.Unsync.GetSource(upStep);
342             }
343             if (firstSource == right)
344             {
345                 return handler(new LinkAddress<LinkIndex>(stepFrom));
346             }
347             return Constants.Continue;
```

```
348         }
349
350         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));
351
352         private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪  left, LinkIndex stepFrom)
353         {
354             var upStep = stepFrom;
355             var firstTarget = Links.Unsync.GetSource(upStep);
356             while (firstTarget != left && firstTarget != upStep)
357             {
358                 upStep = firstTarget;
359                 firstTarget = Links.Unsync.GetTarget(upStep);
360             }
361             if (firstTarget == left)
362             {
363                 return handler(new LinkAddress<LinkIndex>(stepFrom));
364             }
365             return Constants.Continue;
366         }
367
368         #endregion
369
370         #region Update
371
372         public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
373         {
374             var sequence = restrictions.SkipFirst();
375             var newSequence = substitution.SkipFirst();
376
377             if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
378             {
379                 return Constants.Null;
380             }
381             if (sequence.IsNullOrEmpty())
382             {
383                 return Create(substitution);
384             }
385             if (newSequence.IsNullOrEmpty())
386             {
387                 Delete(restrictions);
388                 return Constants.Null;
389             }
390             return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
391             {
392                 ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
393                 Links.EnsureLinkExists(newSequence);
394                 return UpdateCore(sequence, newSequence);
395             }));
396         }
397
398         private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
399         {
400             LinkIndex bestVariant;
401             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
                ↪  !sequence.EqualTo(newSequence))
402             {
403                 bestVariant = CompactCore(newSequence);
404             }
405             else
406             {
407                 bestVariant = CreateCore(newSequence);
408             }
409             // TODO: Check all options only ones before loop execution
410             // Возможно нужно две версии Each, возвращающий фактические последовательности и с
                ↪  маркером,
411             // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
                ↪  можно получить имея только фактические последовательности.
412             foreach (var variant in Each(sequence))
413             {
414                 if (variant != bestVariant)
415                 {
416                     UpdateOneCore(variant, bestVariant);
417                 }
418             }
419             return bestVariant;
```

```
420         }
421
422         private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
423         {
424             if (Options.UseGarbageCollection)
425             {
426                 var sequenceElements = GetSequenceElements(sequence);
427                 var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
428                 var sequenceLink = GetSequenceByElements(sequenceElements);
429                 var newSequenceElements = GetSequenceElements(newSequence);
430                 var newSequenceLink = GetSequenceByElements(newSequenceElements);
431                 if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
432                 {
433                     if (sequenceLink != Constants.Null)
434                     {
435                         Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
436                     }
437                     Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
438                 }
439                 ClearGarbage(sequenceElementsContents.Source);
440                 ClearGarbage(sequenceElementsContents.Target);
441             }
442             else
443             {
444                 if (Options.UseSequenceMarker)
445                 {
446                     var sequenceElements = GetSequenceElements(sequence);
447                     var sequenceLink = GetSequenceByElements(sequenceElements);
448                     var newSequenceElements = GetSequenceElements(newSequence);
449                     var newSequenceLink = GetSequenceByElements(newSequenceElements);
450                     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
451                     {
452                         if (sequenceLink != Constants.Null)
453                         {
454                             Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
455                         }
456                         Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
457                     }
458                 }
459                 else
460                 {
461                     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
462                     {
463                         Links.Unsync.MergeAndDelete(sequence, newSequence);
464                     }
465                 }
466             }
467         }
468
469         #endregion
470
471         #region Delete
472
473         public void Delete(IList<LinkIndex> restrictions)
474         {
475             _sync.ExecuteWriteOperation(() =>
476             {
477                 var sequence = restrictions.SkipFirst();
478                 // TODO: Check all options only ones before loop execution
479                 foreach (var linkToDelete in Each(sequence))
480                 {
481                     DeleteOneCore(linkToDelete);
482                 }
483             });
484         }
485
486         private void DeleteOneCore(LinkIndex link)
487         {
488             if (Options.UseGarbageCollection)
489             {
490                 var sequenceElements = GetSequenceElements(link);
491                 var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
492                 var sequenceLink = GetSequenceByElements(sequenceElements);
493                 if (Options.UseCascadeDelete || CountUsages(link) == 0)
494                 {
495                     if (sequenceLink != Constants.Null)
496                     {
497                         Links.Unsync.Delete(sequenceLink);
```

```
                }
                Links.Unsync.Delete(link);
            }
            ClearGarbage(sequenceElementsContents.Source);
            ClearGarbage(sequenceElementsContents.Target);
        }
        else
        {
            if (Options.UseSequenceMarker)
            {
                var sequenceElements = GetSequenceElements(link);
                var sequenceLink = GetSequenceByElements(sequenceElements);
                if (Options.UseCascadeDelete || CountUsages(link) == 0)
                {
                    if (sequenceLink != Constants.Null)
                    {
                        Links.Unsync.Delete(sequenceLink);
                    }
                    Links.Unsync.Delete(link);
                }
            }
            else
            {
                if (Options.UseCascadeDelete || CountUsages(link) == 0)
                {
                    Links.Unsync.Delete(link);
                }
            }
        }
    }

    #endregion

    #region Compactification

    public void CompactAll()
    {
        _sync.ExecuteWriteOperation(() =>
        {
            var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
            for (int i = 0; i < sequences.Count; i++)
            {
                var sequence = this.ToList(sequences[i]);
                Compact(sequence.ShiftRight());
            }
        });
    }

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

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

    #endregion

    #region Garbage Collection

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

```
575             /// </remarks>
576             [MethodImpl(MethodImplOptions.AggressiveInlining)]
577             private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
                ↪  !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
578
579             private void ClearGarbage(LinkIndex link)
580             {
581                 if (IsGarbage(link))
582                 {
583                     var contents = new Link<ulong>(Links.GetLink(link));
584                     Links.Unsync.Delete(link);
585                     ClearGarbage(contents.Source);
586                     ClearGarbage(contents.Target);
587                 }
588             }
589
590             #endregion
591
592             #region Walkers
593
594             public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
595             {
596                 return _sync.ExecuteReadOperation(() =>
597                 {
598                     var links = Links.Unsync;
599                     foreach (var part in Options.Walker.Walk(sequence))
600                     {
601                         if (!handler(part))
602                         {
603                             return false;
604                         }
605                     }
606                     return true;
607                 });
608             }
609
610             public class Matcher : RightSequenceWalker<LinkIndex>
611             {
612                 private readonly Sequences _sequences;
613                 private readonly IList<LinkIndex> _patternSequence;
614                 private readonly HashSet<LinkIndex> _linksInSequence;
615                 private readonly HashSet<LinkIndex> _results;
616                 private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
617                 private readonly HashSet<LinkIndex> _readAsElements;
618                 private int _filterPosition;
619
620                 public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
                    ↪  HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
                    ↪  HashSet<LinkIndex> readAsElements = null)
621                     : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
622                 {
623                     _sequences = sequences;
624                     _patternSequence = patternSequence;
625                     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                        ↪  Links.Constants.Any && x != ZeroOrMany));
626                     _results = results;
627                     _stopableHandler = stopableHandler;
628                     _readAsElements = readAsElements;
629                 }
630
631                 protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
                    ↪  (_readAsElements != null && _readAsElements.Contains(link)) ||
                    ↪  _linksInSequence.Contains(link);
632
633                 public bool FullMatch(LinkIndex sequenceToMatch)
634                 {
635                     _filterPosition = 0;
636                     foreach (var part in Walk(sequenceToMatch))
637                     {
638                         if (!FullMatchCore(part))
639                         {
640                             break;
641                         }
642                     }
643                     return _filterPosition == _patternSequence.Count;
644                 }
645
646                 private bool FullMatchCore(LinkIndex element)
647                 {
648                     if (_filterPosition == _patternSequence.Count)
```

```
649                 {
650                     _filterPosition = -2; // Длиннее чем нужно
651                     return false;
652                 }
653                 if (_patternSequence[_filterPosition] != Links.Constants.Any
654                  && element != _patternSequence[_filterPosition])
655                 {
656                     _filterPosition = -1;
657                     return false; // Начинается/Продолжается иначе
658                 }
659                 _filterPosition++;
660                 return true;
661             }
662 
663             public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
664             {
665                 var sequenceToMatch = restrictions[Links.Constants.IndexPart];
666                 if (FullMatch(sequenceToMatch))
667                 {
668                     _results.Add(sequenceToMatch);
669                 }
670             }
671 
672             public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
673             {
674                 var sequenceToMatch = restrictions[Links.Constants.IndexPart];
675                 if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
676                 {
677                     return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
678                 }
679                 return Links.Constants.Continue;
680             }
681 
682             public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
683             {
684                 var sequenceToMatch = restrictions[Links.Constants.IndexPart];
685                 var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
686                 if (sequence != Links.Constants.Null && FullMatch(sequenceToMatch) &&
                    ↪ _results.Add(sequenceToMatch))
687                 {
688                     return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
689                 }
690                 return Links.Constants.Continue;
691             }
692 
693             /// <remarks>
694             /// TODO: Add support for LinksConstants.Any
695             /// </remarks>
696             public bool PartialMatch(LinkIndex sequenceToMatch)
697             {
698                 _filterPosition = -1;
699                 foreach (var part in Walk(sequenceToMatch))
700                 {
701                     if (!PartialMatchCore(part))
702                     {
703                         break;
704                     }
705                 }
706                 return _filterPosition == _patternSequence.Count - 1;
707             }
708 
709             private bool PartialMatchCore(LinkIndex element)
710             {
711                 if (_filterPosition == (_patternSequence.Count - 1))
712                 {
713                     return false; // Нашлось
714                 }
715                 if (_filterPosition >= 0)
716                 {
717                     if (element == _patternSequence[_filterPosition + 1])
718                     {
719                         _filterPosition++;
720                     }
721                     else
722                     {
723                         _filterPosition = -1;
724                     }
725                 }
726                 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
    }
}
```

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

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

#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.ShiftRight());
            }
            return sequences.Create(finalSequence.ShiftRight());
        }
```

```
21
22          public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
23          {
24              var list = new List<TLink>();
25              var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
26              sequences.Each(filler.AddSkipFirstAndReturnConstant, new
                ↪  LinkAddress<TLink>(sequence));
27              return list;
28          }
29      }
30  }
```

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

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

1.85 ./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);
    }
}
```

1.86 ./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;
                }
            }
        }
    }
}
```

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

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

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

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

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

        private readonly Func<TLink, bool> _isElement;

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

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

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

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

```
                {
                    var candidate = array[i];
                    if (_equalityComparer.Equals(array[i], default))
                    {
                        continue;
                    }
                    var doubletOffset = i * 2;
                    if (_isElement(candidate))
                    {
                        nextArray[doubletOffset] = candidate;
                    }
                    else
                    {
                        var link = Links.GetLink(candidate);
                        var linkSource = Links.GetSource(link);
                        var linkTarget = Links.GetTarget(link);
                        nextArray[doubletOffset] = linkSource;
                        nextArray[doubletOffset + 1] = linkTarget;
                        if (!hasElements)
                        {
                            hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
                        }
                    }
                }
#if USEARRAYPOOL
                if (array.Length > 1)
                {
                    ArrayPool.Free(array);
                }
#endif
                array = nextArray;
            }
            while (hasElements);
            var filledElementsCount = CountFilledElements(array);
            if (filledElementsCount == array.Length)
            {
                return array;
            }
            else
            {
                return CopyFilledElements(array, filledElementsCount);
            }
        }

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

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

1.88 ./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;
                }
            }
        }
    }
}
```

1.89 ./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);
    }
}
```

1.90 ./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;
        }
```

```
41
42          public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
            ↪  _links.GetOrCreate(GetTop(), element));
43      }
44  }
```

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

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

1.92 ./Platform.Data.Doublets/SynchronizedLinks.cs

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

```
        //}
    }
}
```

1.93 ./Platform.Data.Doublets/UInt64LinksExtensions.cs

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

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

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

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

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

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

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

        public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
        ↪ HashSet<ulong> visited, ulong linkIndex, Func<Link<ulong>, bool> isElement,
        ↪ Action<StringBuilder, Link<ulong>> appendElement, bool renderIndex = false, bool
        ↪ renderDebug = false)
        {
            if (sb == null)
            {
                throw new ArgumentNullException(nameof(sb));
            }
            if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
            ↪ Constants.Itself)
            {
                return;
            }
            if (links.Exists(linkIndex))
            {
                if (visited.Add(linkIndex))
                {
```

```
                    sb.Append('(');
                    var link = new Link<ulong>(links.GetLink(linkIndex));
                    if (renderIndex)
                    {
                        sb.Append(link.Index);
                        sb.Append(':');
                    }
                    if (link.Source == link.Index)
                    {
                        sb.Append(link.Index);
                    }
                    else
                    {
                        var source = new Link<ulong>(links.GetLink(link.Source));
                        if (isElement(source))
                        {
                            appendElement(sb, source);
                        }
                        else
                        {
                            links.AppendStructure(sb, visited, source.Index, isElement,
                                ↪  appendElement, renderIndex);
                        }
                    }
                    sb.Append(' ');
                    if (link.Target == link.Index)
                    {
                        sb.Append(link.Index);
                    }
                    else
                    {
                        var target = new Link<ulong>(links.GetLink(link.Target));
                        if (isElement(target))
                        {
                            appendElement(sb, target);
                        }
                        else
                        {
                            links.AppendStructure(sb, visited, target.Index, isElement,
                                ↪  appendElement, renderIndex);
                        }
                    }
                    sb.Append(')');
                }
                else
                {
                    if (renderDebug)
                    {
                        sb.Append('*');
                    }
                    sb.Append(linkIndex);
                }
            }
            else
            {
                if (renderDebug)
                {
                    sb.Append('~');
                }
                sb.Append(linkIndex);
            }
        }
    }
}
```

1.94 ./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;
```

```
14
15  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17  namespace Platform.Data.Doublets
18  {
19      public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
20      {
21          /// <remarks>
22          /// Альтернативные варианты хранения трансформации (элемента транзакции):
23          ///
24          /// private enum TransitionType
25          /// {
26          ///     Creation,
27          ///     UpdateOf,
28          ///     UpdateTo,
29          ///     Deletion
30          /// }
31          ///
32          /// private struct Transition
33          /// {
34          ///     public ulong TransactionId;
35          ///     public UniqueTimestamp Timestamp;
36          ///     public TransactionItemType Type;
37          ///     public Link Source;
38          ///     public Link Linker;
39          ///     public Link Target;
40          /// }
41          ///
42          /// Или
43          ///
44          /// public struct TransitionHeader
45          /// {
46          ///     public ulong TransactionIdCombined;
47          ///     public ulong TimestampCombined;
48          ///
49          ///     public ulong TransactionId
50          ///     {
51          ///         get
52          ///         {
53          ///             return (ulong) mask &amp;amp; TransactionIdCombined;
54          ///         }
55          ///     }
56          ///
57          ///     public UniqueTimestamp Timestamp
58          ///     {
59          ///         get
60          ///         {
61          ///             return (UniqueTimestamp)mask &amp;amp; TransactionIdCombined;
62          ///         }
63          ///     }
64          ///
65          ///     public TransactionItemType Type
66          ///     {
67          ///         get
68          ///         {
69          ///             // Использовать по одному биту из TransactionId и Timestamp,
70          ///             // для значения в 2 бита, которое представляет тип операции
71          ///             throw new NotImplementedException();
72          ///         }
73          ///     }
74          /// }
75          ///
76          /// private struct Transition
77          /// {
78          ///     public TransitionHeader Header;
79          ///     public Link Source;
80          ///     public Link Linker;
81          ///     public Link Target;
82          /// }
83          ///
84          /// </remarks>
85          public struct Transition : IEquatable<Transition>
86          {
87              public static readonly long Size = Structure<Transition>.Size;
88
89              public readonly ulong TransactionId;
90              public readonly Link<ulong> Before;
91              public readonly Link<ulong> After;
```

```
92              public readonly Timestamp Timestamp;
93
94              public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
                ↪  transactionId, Link<ulong> before, Link<ulong> after)
95              {
96                  TransactionId = transactionId;
97                  Before = before;
98                  After = after;
99                  Timestamp = uniqueTimestampFactory.Create();
100             }
101
102             public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
                ↪  transactionId, Link<ulong> before)
103                 : this(uniqueTimestampFactory, transactionId, before, default)
104             {
105             }
106
107             public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
108                 : this(uniqueTimestampFactory, transactionId, default, default)
109             {
110             }
111
112             public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
                ↪  {After}";
113
114             public override bool Equals(object obj) => obj is Transition transition ?
                ↪  Equals(transition) : false;
115
116             public override int GetHashCode() => (TransactionId, Before, After,
                ↪  Timestamp).GetHashCode();
117
118             public bool Equals(Transition other) => TransactionId == other.TransactionId &&
                ↪  Before == other.Before && After == other.After && Timestamp == other.Timestamp;
119
120             public static bool operator ==(Transition left, Transition right) =>
                ↪  left.Equals(right);
121
122             public static bool operator !=(Transition left, Transition right) => !(left ==
                ↪  right);
123         }
124
125         /// <remarks>
126         /// Другие варианты реализации транзакций (атомарности):
127         ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
                ↪  Target)) и индексов.
128         ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
                ↪  потребуется решить вопрос
129         ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
                ↪  пересечениями идентификаторов.
130         ///
131         /// Где хранить промежуточный список транзакций?
132         ///
133         /// В оперативной памяти:
134         ///  Минусы:
135         ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
136         ///     так как нужно отдельно выделять память под список трансформаций.
137         ///     2. Выделенной оперативной памяти может не хватить, в том случае,
138         ///     если транзакция использует слишком много трансформаций.
139         ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
140         ///         -> Максимальный размер списка трансформаций можно ограничить / задать
                ↪  константой.
141         ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
                ↪  создавая задержку.
142         ///
143         /// На жёстком диске:
144         ///  Минусы:
145         ///     1. Длительный отклик, на запись каждой трансформации.
146         ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
147         ///         -> Это может решаться упаковкой/исключением дублирующих операций.
148         ///         -> Также это может решаться тем, что короткие транзакции вообще
149         ///            не будут записываться в случае отката.
150         ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
                ↪  операции (трансформации)
151         ///        будут записаны в лог.
152         ///
153         /// </remarks>
154         public class Transaction : DisposableBase
155         {
```

```
156             private readonly Queue<Transition> _transitions;
157             private readonly UInt64LinksTransactionsLayer _layer;
158             public bool IsCommitted { get; private set; }
159             public bool IsReverted { get; private set; }
160 
161             public Transaction(UInt64LinksTransactionsLayer layer)
162             {
163                 _layer = layer;
164                 if (_layer._currentTransactionId != 0)
165                 {
166                     throw new NotSupportedException("Nested transactions not supported.");
167                 }
168                 IsCommitted = false;
169                 IsReverted = false;
170                 _transitions = new Queue<Transition>();
171                 SetCurrentTransaction(layer, this);
172             }
173 
174             public void Commit()
175             {
176                 EnsureTransactionAllowsWriteOperations(this);
177                 while (_transitions.Count > 0)
178                 {
179                     var transition = _transitions.Dequeue();
180                     _layer._transitions.Enqueue(transition);
181                 }
182                 _layer._lastCommitedTransactionId = _layer._currentTransactionId;
183                 IsCommitted = true;
184             }
185 
186             private void Revert()
187             {
188                 EnsureTransactionAllowsWriteOperations(this);
189                 var transitionsToRevert = new Transition[_transitions.Count];
190                 _transitions.CopyTo(transitionsToRevert, 0);
191                 for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
192                 {
193                     _layer.RevertTransition(transitionsToRevert[i]);
194                 }
195                 IsReverted = true;
196             }
197 
198             public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
                ↪ Transaction transaction)
199             {
200                 layer._currentTransactionId = layer._lastCommitedTransactionId + 1;
201                 layer._currentTransactionTransitions = transaction._transitions;
202                 layer._currentTransaction = transaction;
203             }
204 
205             public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
206             {
207                 if (transaction.IsReverted)
208                 {
209                     throw new InvalidOperationException("Transation is reverted.");
210                 }
211                 if (transaction.IsCommitted)
212                 {
213                     throw new InvalidOperationException("Transation is commited.");
214                 }
215             }
216 
217             protected override void Dispose(bool manual, bool wasDisposed)
218             {
219                 if (!wasDisposed && _layer != null && !_layer.IsDisposed)
220                 {
221                     if (!IsCommitted && !IsReverted)
222                     {
223                         Revert();
224                     }
225                     _layer.ResetCurrentTransation();
226                 }
227             }
228         }
229 
230         public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
231 
232         private readonly string _logAddress;
233         private readonly FileStream _log;
```

```
234         private readonly Queue<Transition> _transitions;
235         private readonly UniqueTimestampFactory _uniqueTimestampFactory;
236         private Task _transitionsPusher;
237         private Transition _lastCommitedTransition;
238         private ulong _currentTransactionId;
239         private Queue<Transition> _currentTransactionTransitions;
240         private Transaction _currentTransaction;
241         private ulong _lastCommitedTransactionId;
242 
243         public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
244             : base(links)
245         {
246             if (string.IsNullOrWhiteSpace(logAddress))
247             {
248                 throw new ArgumentNullException(nameof(logAddress));
249             }
250             // В первой строке файла хранится последняя закоммиченную транзакцию.
251             // При запуске это используется для проверки удачного закрытия файла лога.
252             // In the first line of the file the last committed transaction is stored.
253             // On startup, this is used to check that the log file is successfully closed.
254             var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
255             var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
256             if (!lastCommitedTransition.Equals(lastWrittenTransition))
257             {
258                 Dispose();
259                 throw new NotSupportedException("Database is damaged, autorecovery is not
                    ↪  supported yet.");
260             }
261             if (lastCommitedTransition == default)
262             {
263                 FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
264             }
265             _lastCommitedTransition = lastCommitedTransition;
266             // TODO: Think about a better way to calculate or store this value
267             var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
268             _lastCommitedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
                ↪  x.TransactionId) : 0;
269             _uniqueTimestampFactory = new UniqueTimestampFactory();
270             _logAddress = logAddress;
271             _log = FileHelpers.Append(logAddress);
272             _transitions = new Queue<Transition>();
273             _transitionsPusher = new Task(TransitionsPusher);
274             _transitionsPusher.Start();
275         }
276 
277         public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
278 
279         public override ulong Create(IList<ulong> restrictions)
280         {
281             var createdLinkIndex = Links.Create();
282             var createdLink = new Link<ulong>(Links.GetLink(createdLinkIndex));
283             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  default, createdLink));
284             return createdLinkIndex;
285         }
286 
287         public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
288         {
289             var linkIndex = restrictions[Constants.IndexPart];
290             var beforeLink = new Link<ulong>(Links.GetLink(linkIndex));
291             linkIndex = Links.Update(restrictions, substitution);
292             var afterLink = new Link<ulong>(Links.GetLink(linkIndex));
293             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  beforeLink, afterLink));
294             return linkIndex;
295         }
296 
297         public override void Delete(IList<ulong> restrictions)
298         {
299             var link = restrictions[Constants.IndexPart];
300             var deletedLink = new Link<ulong>(Links.GetLink(link));
301             Links.Delete(link);
302             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  deletedLink, default));
303         }
304 
305         [MethodImpl(MethodImplOptions.AggressiveInlining)]
306         private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
            ↪  _transitions;
```

```
307
308         private void CommitTransition(Transition transition)
309         {
310             if (_currentTransaction != null)
311             {
312                 Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
313             }
314             var transitions = GetCurrentTransitions();
315             transitions.Enqueue(transition);
316         }
317
318         private void RevertTransition(Transition transition)
319         {
320             if (transition.After.IsNull()) // Revert Deletion with Creation
321             {
322                 Links.Create();
323             }
324             else if (transition.Before.IsNull()) // Revert Creation with Deletion
325             {
326                 Links.Delete(transition.After.Index);
327             }
328             else // Revert Update
329             {
330                 Links.Update(new[] { transition.After.Index, transition.Before.Source,
                    ↪  transition.Before.Target });
331             }
332         }
333
334         private void ResetCurrentTransation()
335         {
336             _currentTransactionId = 0;
337             _currentTransactionTransitions = null;
338             _currentTransaction = null;
339         }
340
341         private void PushTransitions()
342         {
343             if (_log == null || _transitions == null)
344             {
345                 return;
346             }
347             for (var i = 0; i < _transitions.Count; i++)
348             {
349                 var transition = _transitions.Dequeue();
350
351                 _log.Write(transition);
352                 _lastCommitedTransition = transition;
353             }
354         }
355
356         private void TransitionsPusher()
357         {
358             while (!IsDisposed && _transitionsPusher != null)
359             {
360                 Thread.Sleep(DefaultPushDelay);
361                 PushTransitions();
362             }
363         }
364
365         public Transaction BeginTransaction() => new Transaction(this);
366
367         private void DisposeTransitions()
368         {
369             try
370             {
371                 var pusher = _transitionsPusher;
372                 if (pusher != null)
373                 {
374                     _transitionsPusher = null;
375                     pusher.Wait();
376                 }
377                 if (_transitions != null)
378                 {
379                     PushTransitions();
380                 }
381                 _log.DisposeIfPossible();
382                 FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
383             }
384             catch (Exception ex)
```

```
            {
                ex.Ignore();
            }
        }

        #region DisposalBase

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

        #endregion
    }
}
```

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

```
using Platform.Converters;
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);
        }
    }
}
```

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

```
using System.Collections.Generic;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Indexes;

#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);
        }
    }
}
```

1.97 ./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);

```

```
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          public static bool IsCharLink(ulong link) => link <= MapSize;
70
71          public static string FromLinksToString(IList<ulong> linksList)
72          {
73              var sb = new StringBuilder();
74              for (int i = 0; i < linksList.Count; i++)
75              {
76                  sb.Append(FromLinkToChar(linksList[i]));
77              }
78              return sb.ToString();
79          }
80
81          public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
82          {
83              var sb = new StringBuilder();
84              if (links.Exists(link))
85              {
86                  StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
87                      x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                        ↪  element =>
88                      {
89                          sb.Append(FromLinkToChar(element));
90                          return true;
91                      });
92              }
93              return sb.ToString();
94          }
95
96          public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
            ↪  chars.Length);
97
98          public static ulong[] FromCharsToLinkArray(char[] chars, int count)
99          {
100             // char array to ulong array
101             var linksSequence = new ulong[count];
102             for (var i = 0; i < count; i++)
103             {
104                 linksSequence[i] = FromCharToLink(chars[i]);
105             }
106             return linksSequence;
107         }
108
109         public static ulong[] FromStringToLinkArray(string sequence)
110         {
111             // char array to ulong array
112             var linksSequence = new ulong[sequence.Length];
113             for (var i = 0; i < sequence.Length; i++)
114             {
115                 linksSequence[i] = FromCharToLink(sequence[i]);
116             }
117             return linksSequence;
118         }
119
120         public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
121         {
122             var result = new List<ulong[]>();
123             var offset = 0;
124             while (offset < sequence.Length)
125             {
126                 var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
127                 var relativeLength = 1;
128                 var absoluteLength = offset + relativeLength;
129                 while (absoluteLength < sequence.Length &&
130                        currentCategory ==
                           ↪  CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
131                 {
132                     relativeLength++;
133                     absoluteLength++;
134                 }
135                 // char array to ulong array
136                 var innerSequence = new ulong[relativeLength];
137                 var maxLength = offset + relativeLength;
138                 for (var i = offset; i < maxLength; i++)
139                 {
140                     innerSequence[i - offset] = FromCharToLink(sequence[i]);
141                 }
142                 result.Add(innerSequence);
143                 offset += relativeLength;
```

```
            }
            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;
        }
    }
}
```

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Unicode
```

```
{
    public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<TLink, string>
    {
        private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
        private readonly ISequenceWalker<TLink> _sequenceWalker;
        private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;

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

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

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

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

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

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

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

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

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

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

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

        public char Convert(TLink source)
        {
```

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

1.102 ./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}");
        }
    }
}
```

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

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

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

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

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

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

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

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

            ulong x = 10;
            ulong y = 500;

            bool result = false;

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

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

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

            var equalityComparer1 = EqualityComparer<ulong>.Default;

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

            var equalityComparer2 = new UInt64EqualityComparer();

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

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

            var ts6 = Performance.Measure(() =>
```

```
            {
                for (int i = 0; i < N; i++)
                {
                    result = equalityComparer3(x, y);
                }
            });

            var comparer = Comparer<ulong>.Default;

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

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

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

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

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

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

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

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

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

```
    }
}
```

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

```
using Xunit;

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

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

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

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

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

namespace Platform.Data.Doublets.Tests
{
    public static class OptimalVariantSequenceTests
    {
        private static readonly string _sequenceExample = "зеленела зелёная зелень";
        private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
Et malesuada fames ac turpis egestas sed.
Eget velit aliquet sagittis id consectetur purus.
Dignissim cras tincidunt lobortis feugiat vivamus.
Vitae aliquet nec ullamcorper sit.
Lectus quam id leo in vitae.
Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
Integer eget aliquet nibh praesent tristique.
Vitae congue eu consequat ac felis donec et odio.
Tristique et egestas quis ipsum suspendisse.
Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
Imperdiet proin fermentum leo vel orci.
In ante metus dictum at tempor commodo.
Nisi lacus sed viverra tellus in.
Quam vulputate dignissim suspendisse in.
Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
Gravida cum sociis natoque penatibus et magnis dis parturient.
Risus quis varius quam quisque id diam.
Congue nisi vitae suscipit tellus mauris a diam maecenas.
Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
Pharetra vel turpis nunc eget lorem dolor sed viverra.
Mattis pellentesque id nibh tortor id aliquet.
Purus non enim praesent elementum facilisis leo vel.
Etiam sit amet nisl purus in mollis nunc sed.
Tortor at auctor urna nunc id cursus metus aliquam.
```

```
Volutpat odio facilisis mauris sit amet.
Turpis egestas pretium aenean pharetra magna ac placerat.
Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
Porttitor leo a diam sollicitudin tempor id eu.
Volutpat sed cras ornare arcu dui.
Ut aliquam purus sit amet luctus venenatis lectus magna.
Aliquet risus feugiat in ante metus dictum at.
Mattis nunc sed blandit libero.
Elit pellentesque habitant morbi tristique senectus et netus.
Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
Diam donec adipiscing tristique risus nec feugiat.
Pulvinar mattis nunc sed blandit libero volutpat.
Cras fermentum odio eu feugiat pretium nibh ipsum.
In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
A iaculis at erat pellentesque.
Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
Eget lorem dolor sed viverra ipsum nunc.
Leo a diam sollicitudin tempor id eu.
Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";

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

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

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

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

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

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

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

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

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

```
121
122                 var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                    ↪ totalSequenceSymbolFrequencyCounter);
123
124                 var index = new
                    ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
125                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
126
127                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪ linkToItsFrequencyNumberConverter);
128                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪ sequenceToItsLocalElementLevelsConverter);
129
130                 var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                    ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
131
132                 ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                    ↪ index, optimalVariantConverter);
133             }
134         }
135
136         private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
            ↪ SequenceToItsLocalElementLevelsConverter<ulong>
            ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
            ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
137         {
138             index.Add(sequence);
139
140             var optimalVariant = optimalVariantConverter.Convert(sequence);
141
142             var readSequence1 = sequences.ToList(optimalVariant);
143
144             Assert.True(sequence.SequenceEqual(readSequence1));
145         }
146
147         [Fact]
148         public static void SavedSequencesOptimizationTest()
149         {
150             LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
                    ↪ (long.MaxValue + 1UL, ulong.MaxValue));
151
152             using (var memory = new HeapResizableDirectMemory())
153             using (var disposableLinks = new UInt64ResizableDirectMemoryLinks(memory,
                    ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep, constants,
                    ↪ useAvlBasedIndex: false))
154             {
155                 var links = new UInt64Links(disposableLinks);
156
157                 var root = links.CreatePoint();
158
159                 //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
160                 var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
161
162                 var unicodeSymbolMarker = links.GetOrCreate(root,
                    ↪ addressToNumberConverter.Convert(1));
163                 var unicodeSequenceMarker = links.GetOrCreate(root,
                    ↪ addressToNumberConverter.Convert(2));
164
165                 var totalSequenceSymbolFrequencyCounter = new
                    ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
166                 var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                    ↪ totalSequenceSymbolFrequencyCounter);
167                 var index = new
                    ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
168                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
169                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪ linkToItsFrequencyNumberConverter);
170                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪ sequenceToItsLocalElementLevelsConverter);
171
172                 var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
                    ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
173
174                 var unicodeSequencesOptions = new SequencesOptions<ulong>()
```

```
                {
                    UseSequenceMarker = true,
                    SequenceMarkerLink = unicodeSequenceMarker,
                    UseIndex = true,
                    Index = index,
                    LinksToSequenceConverter = optimalVariantConverter,
                    Walker = walker,
                    UseGarbageCollection = true
                };

                var unicodeSequences = new Sequences.Sequences(new
                ↪  SynchronizedLinks<ulong>(links), unicodeSequencesOptions);

                // Create some sequences
                var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
                ↪  StringSplitOptions.RemoveEmptyEntries);
                var arrays = strings.Select(x => x.Select(y =>
                ↪  addressToNumberConverter.Convert(y)).ToArray()).ToArray();
                for (int i = 0; i < arrays.Length; i++)
                {
                    unicodeSequences.Create(arrays[i].ShiftRight());
                }

                var linksCountAfterCreation = links.Count();

                // get list of sequences links
                // for each sequence link
                //   create new sequence version
                //   if new sequence is not the same as sequence link
                //     delete sequence link
                //     collect garbadge
                unicodeSequences.CompactAll();

                var linksCountAfterCompactification = links.Count();

                Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
            }
        }
    }
}
```

1.107 ./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();
```

```
38
39                  var sw3 = Stopwatch.StartNew();
40                  var readSequence2 = new List<ulong>();
41                  SequenceWalker.WalkRight(balancedVariant,
42                                           links.GetSource,
43                                           links.GetTarget,
44                                           links.IsPartialPoint,
45                                           readSequence2.Add);
46                  sw3.Stop();
47
48                  Assert.True(sequence.SequenceEqual(readSequence1));
49
50                  Assert.True(sequence.SequenceEqual(readSequence2));
51
52                  // Assert.True(sw2.Elapsed < sw3.Elapsed);
53
54                  Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
                    ↪  {sw2.Elapsed}");
55
56                  for (var i = 0; i < sequenceLength; i++)
57                  {
58                      links.Delete(sequence[i]);
59                  }
60              }
61          }
62      }
63  }
```

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

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

```
                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);
        }
    }
}
```

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

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

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);
            }
        }
    }
}
```

1.110 ./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.Collections.Arrays;
using Platform.Random;
using Platform.IO;
using Platform.Singletons;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Unicode;

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

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

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

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

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

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

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

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

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

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

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

        //    const long sequenceLength = 8;

        //    const ulong itself = LinksConstants.Itself;

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

```

```
79          //        SequencesOptions o = new SequencesOptions();
80
81          // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
82          //        o.
83
84          //        var sequences = new Sequences(links);
85
86          //        var sw1 = Stopwatch.StartNew();
87          //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
88
89          //        var sw2 = Stopwatch.StartNew();
90          //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
91
92          //        Assert.True(results1.Count > results2.Length);
93          //        Assert.True(sw1.Elapsed > sw2.Elapsed);
94
95          //        for (var i = 0; i < sequenceLength; i++)
96          //            links.Delete(sequence[i]);
97          //    }
98
99          //    File.Delete(tempFilename);
100         //}
101
102         [Fact]
103         public static void AllVariantsSearchTest()
104         {
105             const long sequenceLength = 8;
106
107             using (var scope = new TempLinksTestScope(useSequences: true))
108             {
109                 var links = scope.Links;
110                 var sequences = scope.Sequences;
111
112                 var sequence = new ulong[sequenceLength];
113                 for (var i = 0; i < sequenceLength; i++)
114                 {
115                     sequence[i] = links.Create();
116                 }
117
118                 var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119
120                 //for (int i = 0; i < createResults.Length; i++)
121                 //    sequences.Create(createResults[i]);
122
123                 var sw0 = Stopwatch.StartNew();
124                 var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125
126                 var sw1 = Stopwatch.StartNew();
127                 var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128
129                 var sw2 = Stopwatch.StartNew();
130                 var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131
132                 var sw3 = Stopwatch.StartNew();
133                 var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
134
135                 var intersection0 = createResults.Intersect(searchResults0).ToList();
136                 Assert.True(intersection0.Count == searchResults0.Count);
137                 Assert.True(intersection0.Count == createResults.Length);
138
139                 var intersection1 = createResults.Intersect(searchResults1).ToList();
140                 Assert.True(intersection1.Count == searchResults1.Count);
141                 Assert.True(intersection1.Count == createResults.Length);
142
143                 var intersection2 = createResults.Intersect(searchResults2).ToList();
144                 Assert.True(intersection2.Count == searchResults2.Count);
145                 Assert.True(intersection2.Count == createResults.Length);
146
147                 var intersection3 = createResults.Intersect(searchResults3).ToList();
148                 Assert.True(intersection3.Count == searchResults3.Count);
149                 Assert.True(intersection3.Count == createResults.Length);
150
151                 for (var i = 0; i < sequenceLength; i++)
152                 {
153                     links.Delete(sequence[i]);
154                 }
155             }
156         }
157
158         [Fact]
```

```
159         public static void BalancedVariantSearchTest()
160         {
161             const long sequenceLength = 200;
162
163             using (var scope = new TempLinksTestScope(useSequences: true))
164             {
165                 var links = scope.Links;
166                 var sequences = scope.Sequences;
167
168                 var sequence = new ulong[sequenceLength];
169                 for (var i = 0; i < sequenceLength; i++)
170                 {
171                     sequence[i] = links.Create();
172                 }
173
174                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
175
176                 var sw1 = Stopwatch.StartNew();
177                 var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
178
179                 var sw2 = Stopwatch.StartNew();
180                 var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
181
182                 var sw3 = Stopwatch.StartNew();
183                 var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
184
185                 // На количестве в 200 элементов это будет занимать вечность
186                 //var sw4 = Stopwatch.StartNew();
187                 //var searchResults4 = sequences.Each(sequence); sw4.Stop();
188
189                 Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
190
191                 Assert.True(searchResults3.Count == 1 && balancedVariant ==
                    ↪ searchResults3.First());
192
193                 //Assert.True(sw1.Elapsed < sw2.Elapsed);
194
195                 for (var i = 0; i < sequenceLength; i++)
196                 {
197                     links.Delete(sequence[i]);
198                 }
199             }
200         }
201
202         [Fact]
203         public static void AllPartialVariantsSearchTest()
204         {
205             const long sequenceLength = 8;
206
207             using (var scope = new TempLinksTestScope(useSequences: true))
208             {
209                 var links = scope.Links;
210                 var sequences = scope.Sequences;
211
212                 var sequence = new ulong[sequenceLength];
213                 for (var i = 0; i < sequenceLength; i++)
214                 {
215                     sequence[i] = links.Create();
216                 }
217
218                 var createResults = sequences.CreateAllVariants2(sequence);
219
220                 //var createResultsStrings = createResults.Select(x => x + ": " +
                    ↪ sequences.FormatSequence(x)).ToList();
221                 //Global.Trash = createResultsStrings;
222
223                 var partialSequence = new ulong[sequenceLength - 2];
224
225                 Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
226
227                 var sw1 = Stopwatch.StartNew();
228                 var searchResults1 =
                    ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
229
230                 var sw2 = Stopwatch.StartNew();
231                 var searchResults2 =
                    ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
232
233                 //var sw3 = Stopwatch.StartNew();
```

```
234                 //var searchResults3 =
                    ↪  sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
235
236                 var sw4 = Stopwatch.StartNew();
237                 var searchResults4 =
                    ↪  sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
238
239                 //Global.Trash = searchResults3;
240
241                 //var searchResults1Strings = searchResults1.Select(x => x + ": " +
                    ↪  sequences.FormatSequence(x)).ToList();
242                 //Global.Trash = searchResults1Strings;
243
244                 var intersection1 = createResults.Intersect(searchResults1).ToList();
245                 Assert.True(intersection1.Count == createResults.Length);
246
247                 var intersection2 = createResults.Intersect(searchResults2).ToList();
248                 Assert.True(intersection2.Count == createResults.Length);
249
250                 var intersection4 = createResults.Intersect(searchResults4).ToList();
251                 Assert.True(intersection4.Count == createResults.Length);
252
253                 for (var i = 0; i < sequenceLength; i++)
254                 {
255                     links.Delete(sequence[i]);
256                 }
257             }
258         }
259
260         [Fact]
261         public static void BalancedPartialVariantsSearchTest()
262         {
263             const long sequenceLength = 200;
264
265             using (var scope = new TempLinksTestScope(useSequences: true))
266             {
267                 var links = scope.Links;
268                 var sequences = scope.Sequences;
269
270                 var sequence = new ulong[sequenceLength];
271                 for (var i = 0; i < sequenceLength; i++)
272                 {
273                     sequence[i] = links.Create();
274                 }
275
276                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
277
278                 var balancedVariant = balancedVariantConverter.Convert(sequence);
279
280                 var partialSequence = new ulong[sequenceLength - 2];
281
282                 Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
283
284                 var sw1 = Stopwatch.StartNew();
285                 var searchResults1 =
                    ↪  sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
286
287                 var sw2 = Stopwatch.StartNew();
288                 var searchResults2 =
                    ↪  sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
289
290                 Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
291
292                 Assert.True(searchResults2.Count == 1 && balancedVariant ==
                    ↪  searchResults2.First());
293
294                 for (var i = 0; i < sequenceLength; i++)
295                 {
296                     links.Delete(sequence[i]);
297                 }
298             }
299         }
300
301         [Fact(Skip = "Correct implementation is pending")]
302         public static void PatternMatchTest()
303         {
304             var zeroOrMany = Sequences.Sequences.ZeroOrMany;
305
306             using (var scope = new TempLinksTestScope(useSequences: true))
```

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

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

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

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

                var balancedVariant = balancedVariantConverter.Convert(sequence);

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

                var doublet = links.GetSource(balancedVariant);

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

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

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

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

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

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

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

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

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

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

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

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

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

                index.Add(sequence);

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

        /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
         ↪  D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
         ↪  %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
        private static readonly string _exampleText =
            @"([english
             ↪  version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
```

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

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?

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

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?

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?

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?

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?

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?

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

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

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

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

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

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

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

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

...

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

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

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

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

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

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

```

```
460                 var compressedVariant = compressingConverter.Convert(sequence);
461
462                 // 1: [1]       (1->1) point
463                 // 2: [2]       (2->2) point
464                 // 3: [1,2]     (1->2) doublet
465                 // 4: [1,2,1,2] (3->3) doublet
466
467                 Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
468                 Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
469                 Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
470                 Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
471
472                 var source = _constants.SourcePart;
473                 var target = _constants.TargetPart;
474
475                 Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
476                 Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
477                 Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
478                 Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
479
480                 // 4 - length of sequence
481                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
                    ↪  == sequence[0]);
482                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
                    ↪  == sequence[1]);
483                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
                    ↪  == sequence[2]);
484                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
                    ↪  == sequence[3]);
485             }
486         }
487
488         [Fact]
489         public static void CompressionEfficiencyTest()
490         {
491             var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
                ↪  StringSplitOptions.RemoveEmptyEntries);
492             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
493             var totalCharacters = arrays.Select(x => x.Length).Sum();
494
495             using (var scope1 = new TempLinksTestScope(useSequences: true))
496             using (var scope2 = new TempLinksTestScope(useSequences: true))
497             using (var scope3 = new TempLinksTestScope(useSequences: true))
498             {
499                 scope1.Links.Unsync.UseUnicode();
500                 scope2.Links.Unsync.UseUnicode();
501                 scope3.Links.Unsync.UseUnicode();
502
503                 var balancedVariantConverter1 = new
                    ↪  BalancedVariantConverter<ulong>(scope1.Links.Unsync);
504                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
505                 var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
506                 var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
                    ↪  balancedVariantConverter1, linkFrequenciesCache1,
                    ↪  doInitialFrequenciesIncrement: false);
507
508                 //var compressor2 = scope2.Sequences;
509                 var compressor3 = scope3.Sequences;
510
511                 var constants = Default<LinksConstants<ulong>>.Instance;
512
513                 var sequences = compressor3;
514                 //var meaningRoot = links.CreatePoint();
515                 //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
516                 //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
517                 //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                    ↪  constants.Itself);
518
519                 //var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
520                 //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
                    ↪  unaryOne);
521                 //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
522                 //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
```

```
523                 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
524                 //var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
525
526                 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
527
528                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache3);
529
530                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
                    ↪  linkToItsFrequencyNumberConverter);
531                 var optimalVariantConverter = new
                    ↪  OptimalVariantConverter<ulong>(scope3.Links.Unsync,
                    ↪  sequenceToItsLocalElementLevelsConverter);
532
533                 var compressed1 = new ulong[arrays.Length];
534                 var compressed2 = new ulong[arrays.Length];
535                 var compressed3 = new ulong[arrays.Length];
536
537                 var START = 0;
538                 var END = arrays.Length;
539
540                 //for (int i = START; i < END; i++)
541                 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
542
543                 var initialCount1 = scope2.Links.Unsync.Count();
544
545                 var sw1 = Stopwatch.StartNew();
546
547                 for (int i = START; i < END; i++)
548                 {
549                     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
550                     compressed1[i] = compressor1.Convert(arrays[i]);
551                 }
552
553                 var elapsed1 = sw1.Elapsed;
554
555                 var balancedVariantConverter2 = new
                    ↪  BalancedVariantConverter<ulong>(scope2.Links.Unsync);
556
557                 var initialCount2 = scope2.Links.Unsync.Count();
558
559                 var sw2 = Stopwatch.StartNew();
560
561                 for (int i = START; i < END; i++)
562                 {
563                     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
564                 }
565
566                 var elapsed2 = sw2.Elapsed;
567
568                 for (int i = START; i < END; i++)
569                 {
570                     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
571                 }
572
573                 var initialCount3 = scope3.Links.Unsync.Count();
574
575                 var sw3 = Stopwatch.StartNew();
576
577                 for (int i = START; i < END; i++)
578                 {
579                     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
580                     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
581                 }
582
583                 var elapsed3 = sw3.Elapsed;
584
585                 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
                    ↪  Optimal variant: {elapsed3}");
586
587                 // Assert.True(elapsed1 > elapsed2);
588
589                 // Checks
590                 for (int i = START; i < END; i++)
```

```
591                 {
592                     var sequence1 = compressed1[i];
593                     var sequence2 = compressed2[i];
594                     var sequence3 = compressed3[i];
595 
596                     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪  scope1.Links.Unsync);
597 
598                     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪  scope2.Links.Unsync);
599 
600                     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
                        ↪  scope3.Links.Unsync);
601 
602                     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
                        ↪  link.IsPartialPoint());
603                     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
                        ↪  link.IsPartialPoint());
604                     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
                        ↪  link.IsPartialPoint());
605 
606                     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
607                     //    Assert.False(structure1 == structure2);
608                     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
609                     //    Assert.False(structure3 == structure2);
610 
611                     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
612                     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
613                 }
614 
615                 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
                    ↪  totalCharacters);
616                 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
                    ↪  totalCharacters);
617                 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
                    ↪  totalCharacters);
618 
619                 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
                    ↪  totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
                    ↪  totalCharacters}");
620 
621                 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
                    ↪  scope2.Links.Unsync.Count() - initialCount2);
622                 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
                    ↪  scope2.Links.Unsync.Count() - initialCount2);
623 
624                 var duplicateProvider1 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
625                 var duplicateProvider2 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
626                 var duplicateProvider3 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
627 
628                 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
629                 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
630                 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
631 
632                 var duplicates1 = duplicateCounter1.Count();
633 
634                 ConsoleHelpers.Debug("------");
635 
636                 var duplicates2 = duplicateCounter2.Count();
637 
638                 ConsoleHelpers.Debug("------");
639 
640                 var duplicates3 = duplicateCounter3.Count();
641 
642                 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
643 
644                 linkFrequenciesCache1.ValidateFrequencies();
645                 linkFrequenciesCache3.ValidateFrequencies();
646             }
647         }
648 
649         [Fact]
```

```
650         public static void CompressionStabilityTest()
651         {
652             // TODO: Fix bug (do a separate test)
653             //const ulong minNumbers = 0;
654             //const ulong maxNumbers = 1000;
655 
656             const ulong minNumbers = 10000;
657             const ulong maxNumbers = 12500;
658 
659             var strings = new List<string>();
660 
661             for (ulong i = minNumbers; i < maxNumbers; i++)
662             {
663                 strings.Add(i.ToString());
664             }
665 
666             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
667             var totalCharacters = arrays.Select(x => x.Length).Sum();
668 
669             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                ↪  SequencesOptions<ulong> { UseCompression = true,
                ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
670             using (var scope2 = new TempLinksTestScope(useSequences: true))
671             {
672                 scope1.Links.UseUnicode();
673                 scope2.Links.UseUnicode();
674 
675                 //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
676                 var compressor1 = scope1.Sequences;
677                 var compressor2 = scope2.Sequences;
678 
679                 var compressed1 = new ulong[arrays.Length];
680                 var compressed2 = new ulong[arrays.Length];
681 
682                 var sw1 = Stopwatch.StartNew();
683 
684                 var START = 0;
685                 var END = arrays.Length;
686 
687                 // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
688                 // Stability issue starts at 10001 or 11000
689                 //for (int i = START; i < END; i++)
690                 //{
691                 //    var first = compressor1.Compress(arrays[i]);
692                 //    var second = compressor1.Compress(arrays[i]);
693 
694                 //    if (first == second)
695                 //        compressed1[i] = first;
696                 //    else
697                 //    {
698                 //        // TODO: Find a solution for this case
699                 //    }
700                 //}
701 
702                 for (int i = START; i < END; i++)
703                 {
704                     var first = compressor1.Create(arrays[i].ShiftRight());
705                     var second = compressor1.Create(arrays[i].ShiftRight());
706 
707                     if (first == second)
708                     {
709                         compressed1[i] = first;
710                     }
711                     else
712                     {
713                         // TODO: Find a solution for this case
714                     }
715                 }
716 
717                 var elapsed1 = sw1.Elapsed;
718 
719                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
720 
721                 var sw2 = Stopwatch.StartNew();
722 
723                 for (int i = START; i < END; i++)
724                 {
725                     var first = balancedVariantConverter.Convert(arrays[i]);
726                     var second = balancedVariantConverter.Convert(arrays[i]);
727 
```

```
728                     if (first == second)
729                     {
730                         compressed2[i] = first;
731                     }
732                 }
733 
734                 var elapsed2 = sw2.Elapsed;
735 
736                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪ {elapsed2}");
737 
738                 Assert.True(elapsed1 > elapsed2);
739 
740                 // Checks
741                 for (int i = START; i < END; i++)
742                 {
743                     var sequence1 = compressed1[i];
744                     var sequence2 = compressed2[i];
745 
746                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
747                     {
748                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪ scope1.Links);
749 
750                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪ scope2.Links);
751 
752                         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                            ↪ link.IsPartialPoint());
753                         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                            ↪ link.IsPartialPoint());
754 
755                         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                            ↪ arrays[i].Length > 3)
756                         //    Assert.False(structure1 == structure2);
757 
758                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
759                     }
760                 }
761 
762                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
763                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
764 
765                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪ totalCharacters}");
766 
767                 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
768 
769                 //compressor1.ValidateFrequencies();
770             }
771         }
772 
773         [Fact]
774         public static void RundomNumbersCompressionQualityTest()
775         {
776             const ulong N = 500;
777 
778             //const ulong minNumbers = 10000;
779             //const ulong maxNumbers = 20000;
780 
781             //var strings = new List<string>();
782 
783             //for (ulong i = 0; i < N; i++)
784             //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
                ↪ maxNumbers).ToString());
785 
786             var strings = new List<string>();
787 
788             for (ulong i = 0; i < N; i++)
789             {
790                 strings.Add(RandomHelpers.Default.NextUInt64().ToString());
791             }
792 
793             strings = strings.Distinct().ToList();
794 
795             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
796             var totalCharacters = arrays.Select(x => x.Length).Sum();
797 
```

```
798             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                ↪  SequencesOptions<ulong> { UseCompression = true,
                ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
799             using (var scope2 = new TempLinksTestScope(useSequences: true))
800             {
801                 scope1.Links.UseUnicode();
802                 scope2.Links.UseUnicode();
803
804                 var compressor1 = scope1.Sequences;
805                 var compressor2 = scope2.Sequences;
806
807                 var compressed1 = new ulong[arrays.Length];
808                 var compressed2 = new ulong[arrays.Length];
809
810                 var sw1 = Stopwatch.StartNew();
811
812                 var START = 0;
813                 var END = arrays.Length;
814
815                 for (int i = START; i < END; i++)
816                 {
817                     compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
818                 }
819
820                 var elapsed1 = sw1.Elapsed;
821
822                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
823
824                 var sw2 = Stopwatch.StartNew();
825
826                 for (int i = START; i < END; i++)
827                 {
828                     compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
829                 }
830
831                 var elapsed2 = sw2.Elapsed;
832
833                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪  {elapsed2}");
834
835                 Assert.True(elapsed1 > elapsed2);
836
837                 // Checks
838                 for (int i = START; i < END; i++)
839                 {
840                     var sequence1 = compressed1[i];
841                     var sequence2 = compressed2[i];
842
843                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
844                     {
845                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪  scope1.Links);
846
847                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪  scope2.Links);
848
849                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
850                     }
851                 }
852
853                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
854                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
855
856                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters}");
857
858                 // Can be worse than balanced variant
859                 //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
860
861                 //compressor1.ValidateFrequencies();
862             }
863         }
864
865         [Fact]
866         public static void AllTreeBreakDownAtSequencesCreationBugTest()
867         {
868             // Made out of AllPossibleConnectionsTest test.
869
```

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

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

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

                var createResults = sequences.CreateAllVariants2(sequence);

                Global.Trash = createResults;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

                for (var i = 0; i < sequenceLength; i++)
                {
```

```
                    links.Delete(sequence[i]);
                }
            }
        }

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

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

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

                var createResults = sequences.CreateAllVariants2(sequence);

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

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

                    sequences.CalculateAllUsages(linksTotalUsages1);

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

                    sequences.CalculateAllUsages2(linksTotalUsages2);

                    var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
                    Assert.True(intersection1.Count == linksTotalUsages2.Length);
                }

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

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

```
using System.IO;
using Platform.Disposables;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.ResizableDirectMemory.Specific;

namespace Platform.Data.Doublets.Tests
{
    public class TempLinksTestScope : DisposableBase
    {
        public ILinks<ulong> MemoryAdapter { get; }
        public SynchronizedLinks<ulong> Links { get; }
        public Sequences.Sequences Sequences { get; }
        public string TempFilename { get; }
        public string TempTransactionLogFilename { get; }
        private readonly bool _deleteFiles;

        public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
        ↪  useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
        ↪  useLog) { }

        public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
        ↪  true, bool useSequences = false, bool useLog = false)
        {
            _deleteFiles = deleteFiles;
            TempFilename = Path.GetTempFileName();
            TempTransactionLogFilename = Path.GetTempFileName();
            var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
```

```
            MemoryAdapter = useLog ? (ILinks<ulong>)new
            ↪  UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
            ↪  coreMemoryAdapter;
            Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
            if (useSequences)
            {
                Sequences = new Sequences.Sequences(Links, sequencesOptions);
            }
        }

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

        public void DeleteFiles()
        {
            File.Delete(TempFilename);
            File.Delete(TempTransactionLogFilename);
        }
    }
}
```

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

```
using System.Collections.Generic;
using Xunit;
using Platform.Ranges;
using Platform.Numbers;
using Platform.Random;
using Platform.Setters;

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

            var equalityComparer = EqualityComparer<T>.Default;

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

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

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

            var linkAddress = links.Create();

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

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

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

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

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

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

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

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

```
50
51              // Update link to reference null (prepare for delete)
52              var updated = links.Update(linkAddress, constants.Null, constants.Null);
53
54              Assert.True(equalityComparer.Equals(updated, linkAddress));
55
56              link = new Link<T>(links.GetLink(linkAddress));
57
58              Assert.True(equalityComparer.Equals(link.Source, constants.Null));
59              Assert.True(equalityComparer.Equals(link.Target, constants.Null));
60
61              // Delete link
62              links.Delete(linkAddress);
63
64              Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
65
66              setter = new Setter<T>(constants.Null);
67              links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
68
69              Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
70          }
71
72          public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
73          {
74              // Constants
75              var constants = links.Constants;
76              var equalityComparer = EqualityComparer<T>.Default;
77
78              var h106E = new Hybrid<T>(106L, isExternal: true);
79              var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
80              var h108E = new Hybrid<T>(-108L);
81
82              Assert.Equal(106L, h106E.AbsoluteValue);
83              Assert.Equal(107L, h107E.AbsoluteValue);
84              Assert.Equal(108L, h108E.AbsoluteValue);
85
86              // Create Link (External -> External)
87              var linkAddress1 = links.Create();
88
89              links.Update(linkAddress1, h106E, h108E);
90
91              var link1 = new Link<T>(links.GetLink(linkAddress1));
92
93              Assert.True(equalityComparer.Equals(link1.Source, h106E));
94              Assert.True(equalityComparer.Equals(link1.Target, h108E));
95
96              // Create Link (Internal -> External)
97              var linkAddress2 = links.Create();
98
99              links.Update(linkAddress2, linkAddress1, h108E);
100
101             var link2 = new Link<T>(links.GetLink(linkAddress2));
102
103             Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
104             Assert.True(equalityComparer.Equals(link2.Target, h108E));
105
106             // Create Link (Internal -> Internal)
107             var linkAddress3 = links.Create();
108
109             links.Update(linkAddress3, linkAddress1, linkAddress2);
110
111             var link3 = new Link<T>(links.GetLink(linkAddress3));
112
113             Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
114             Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
115
116             // Search for created link
117             var setter1 = new Setter<T>(constants.Null);
118             links.Each(h106E, h108E, setter1.SetAndReturnFalse);
119
120             Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
121
122             // Search for nonexistent link
123             var setter2 = new Setter<T>(constants.Null);
124             links.Each(h106E, h107E, setter2.SetAndReturnFalse);
125
126             Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
127
128             // Update link to reference null (prepare for delete)
129             var updated = links.Update(linkAddress3, constants.Null, constants.Null);
```

```
130
131             Assert.True(equalityComparer.Equals(updated, linkAddress3));
132
133             link3 = new Link<T>(links.GetLink(linkAddress3));
134
135             Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
136             Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
137
138             // Delete link
139             links.Delete(linkAddress3);
140
141             Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));
142
143             var setter3 = new Setter<T>(constants.Null);
144             links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
145
146             Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
147         }
148
149         public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
            ↪ links, int maximumOperationsPerCycle)
150         {
151             var comparer = Comparer<TLink>.Default;
152             for (var N = 1; N < maximumOperationsPerCycle; N++)
153             {
154                 var random = new System.Random(N);
155                 var created = 0;
156                 var deleted = 0;
157                 for (var i = 0; i < N; i++)
158                 {
159                     long linksCount = (Integer<TLink>)links.Count();
160                     var createPoint = random.NextBoolean();
161                     if (linksCount > 2 && createPoint)
162                     {
163                         var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
164                         TLink source = (Integer<TLink>)random.NextUInt64(linksAddressRange);
165                         TLink target = (Integer<TLink>)random.NextUInt64(linksAddressRange);
                            ↪ //-V3086
166                         var resultLink = links.GetOrCreate(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 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.IO;
5  using System.Text;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Xunit;
9  using Platform.Disposables;
10 using Platform.Ranges;
11 using Platform.Random;
12 using Platform.Timestamps;
13 using Platform.Reflection;
```

```
using Platform.Singletons;
using Platform.Scopes;
using Platform.Counters;
using Platform.Diagnostics;
using Platform.IO;
using Platform.Memory;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.ResizableDirectMemory.Specific;

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

        private const long Iterations = 10 * 1024;

        #region Concept

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

        [Fact]
        public static void CascadeUpdateTest()
        {
            var itself = _constants.Itself;
            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;

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

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

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

                l2 = links.Update(l2, l1);

                links.Delete(l2);

                Global.Trash = links.Count();

                links.Unsync.DisposeIfPossible(); // Close links to access log

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

        [Fact]
        public static void BasicTransactionLogTest()
        {
            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;
                var l1 = links.Create();
                var l2 = links.Create();

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

                links.Delete(l1);

                links.Unsync.DisposeIfPossible(); // Close links to access log

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

```

```
89          [Fact]
90          public static void TransactionAutoRevertedTest()
91          {
92              // Auto Reverted (Because no commit at transaction)
93              using (var scope = new TempLinksTestScope(useLog: true))
94              {
95                  var links = scope.Links;
96                  var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
97                  using (var transaction = transactionsLayer.BeginTransaction())
98                  {
99                      var l1 = links.Create();
100                     var l2 = links.Create();
101
102                     links.Update(l2, l2, l1, l2);
103                 }
104
105                 Assert.Equal(0UL, links.Count());
106
107                 links.Unsync.DisposeIfPossible();
108
109                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
                    ↪ cope.TempTransactionLogFilename);
110                 Assert.Single(transitions);
111             }
112         }
113
114         [Fact]
115         public static void TransactionUserCodeErrorNoDataSavedTest()
116         {
117             // User Code Error (Autoreverted), no data saved
118             var itself = _constants.Itself;
119
120             TempLinksTestScope lastScope = null;
121             try
122             {
123                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪ useLog: true))
124                 {
125                     var links = scope.Links;
126                     var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
                        ↪ atorBase<ulong>)links.Unsync).Links;
127                     using (var transaction = transactionsLayer.BeginTransaction())
128                     {
129                         var l1 = links.CreateAndUpdate(itself, itself);
130                         var l2 = links.CreateAndUpdate(itself, itself);
131
132                         l2 = links.Update(l2, l2, l1, l2);
133
134                         links.CreateAndUpdate(l2, itself);
135                         links.CreateAndUpdate(l2, itself);
136
137                         //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
                            ↪ tion>(scope.TempTransactionLogFilename);
138
139                         l2 = links.Update(l2, l1);
140
141                         links.Delete(l2);
142
143                         ExceptionThrower();
144
145                         transaction.Commit();
146                     }
147
148                     Global.Trash = links.Count();
149                 }
150             }
151             catch
152             {
153                 Assert.False(lastScope == null);
154
155                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
                    ↪ astScope.TempTransactionLogFilename);
156
157                 Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
                    ↪ transitions[0].After.IsNull());
158
159                 lastScope.DeleteFiles();
160             }
161         }
```

```
162
163         [Fact]
164         public static void TransactionUserCodeErrorSomeDataSavedTest()
165         {
166             // User Code Error (Autoreverted), some data saved
167             var itself = _constants.Itself;
168
169             TempLinksTestScope lastScope = null;
170             try
171             {
172                 ulong l1;
173                 ulong l2;
174
175                 using (var scope = new TempLinksTestScope(useLog: true))
176                 {
177                     var links = scope.Links;
178                     l1 = links.CreateAndUpdate(itself, itself);
179                     l2 = links.CreateAndUpdate(itself, itself);
180
181                     l2 = links.Update(l2, l2, l1, l2);
182
183                     links.CreateAndUpdate(l2, itself);
184                     links.CreateAndUpdate(l2, itself);
185
186                     links.Unsync.DisposeIfPossible();
187
188                     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
                        ↪ scope.TempTransactionLogFilename);
189                 }
190
191                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪ useLog: true))
192                 {
193                     var links = scope.Links;
194                     var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
195                     using (var transaction = transactionsLayer.BeginTransaction())
196                     {
197                         l2 = links.Update(l2, l1);
198
199                         links.Delete(l2);
200
201                         ExceptionThrower();
202
203                         transaction.Commit();
204                     }
205
206                     Global.Trash = links.Count();
207                 }
208             }
209             catch
210             {
211                 Assert.False(lastScope == null);
212
213                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
                    ↪ Scope.TempTransactionLogFilename);
214
215                 lastScope.DeleteFiles();
216             }
217         }
218
219         [Fact]
220         public static void TransactionCommit()
221         {
222             var itself = _constants.Itself;
223
224             var tempDatabaseFilename = Path.GetTempFileName();
225             var tempTransactionLogFilename = Path.GetTempFileName();
226
227             // Commit
228             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                ↪ tempTransactionLogFilename))
229             using (var links = new UInt64Links(memoryAdapter))
230             {
231                 using (var transaction = memoryAdapter.BeginTransaction())
232                 {
233                     var l1 = links.CreateAndUpdate(itself, itself);
234                     var l2 = links.CreateAndUpdate(itself, itself);
235
```

```
236                     Global.Trash = links.Update(l2, l2, l1, l2);
237
238                     links.Delete(l1);
239
240                     transaction.Commit();
241                 }
242
243                 Global.Trash = links.Count();
244             }
245
246             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
247         }
248
249         [Fact]
250         public static void TransactionDamage()
251         {
252             var itself = _constants.Itself;
253
254             var tempDatabaseFilename = Path.GetTempFileName();
255             var tempTransactionLogFilename = Path.GetTempFileName();
256
257             // Commit
258             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                ↪ tempTransactionLogFilename))
259             using (var links = new UInt64Links(memoryAdapter))
260             {
261                 using (var transaction = memoryAdapter.BeginTransaction())
262                 {
263                     var l1 = links.CreateAndUpdate(itself, itself);
264                     var l2 = links.CreateAndUpdate(itself, itself);
265
266                     Global.Trash = links.Update(l2, l2, l1, l2);
267
268                     links.Delete(l1);
269
270                     transaction.Commit();
271                 }
272
273                 Global.Trash = links.Count();
274             }
275
276             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
277
278             // Damage database
279
280             FileHelpers.WriteFirst(tempTransactionLogFilename, new
                ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
281
282             // Try load damaged database
283             try
284             {
285                 // TODO: Fix
286                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                    ↪ tempTransactionLogFilename))
287                 using (var links = new UInt64Links(memoryAdapter))
288                 {
289                     Global.Trash = links.Count();
290                 }
291             }
292             catch (NotSupportedException ex)
293             {
294                 Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
                    ↪ yet.");
295             }
296
297             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
298
299             File.Delete(tempDatabaseFilename);
300             File.Delete(tempTransactionLogFilename);
301         }
302
303         [Fact]
304         public static void Bug1Test()
305         {
```

```
306             var tempDatabaseFilename = Path.GetTempFileName();
307             var tempTransactionLogFilename = Path.GetTempFileName();
308 
309             var itself = _constants.Itself;
310 
311             // User Code Error (Autoreverted), some data saved
312             try
313             {
314                 ulong l1;
315                 ulong l2;
316 
317                 using (var memory = new UInt64ResizableDirectMemoryLinks(tempDatabaseFilename))
318                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
                    ↪ 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 memory = new UInt64ResizableDirectMemoryLinks(tempDatabaseFilename))
333                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
                    ↪ tempTransactionLogFilename))
334                 using (var links = new UInt64Links(memoryAdapter))
335                 {
336                     using (var transaction = memoryAdapter.BeginTransaction())
337                     {
338                         l2 = links.Update(l2, l1);
339 
340                         links.Delete(l2);
341 
342                         ExceptionThrower();
343 
344                         transaction.Commit();
345                     }
346 
347                     Global.Trash = links.Count();
348                 }
349             }
350             catch
351             {
352                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                    ↪ TransactionLogFilename);
353             }
354 
355             File.Delete(tempDatabaseFilename);
356             File.Delete(tempTransactionLogFilename);
357         }
358 
359         private static void ExceptionThrower() => throw new InvalidOperationException();
360 
361         [Fact]
362         public static void PathsTest()
363         {
364             var source = _constants.SourcePart;
365             var target = _constants.TargetPart;
366 
367             using (var scope = new TempLinksTestScope())
368             {
369                 var links = scope.Links;
370                 var l1 = links.CreatePoint();
371                 var l2 = links.CreatePoint();
372 
373                 var r1 = links.GetByKeys(l1, source, target, source);
374                 var r2 = links.CheckPathExistance(l2, l2, l2, l2);
375             }
376         }
377 
378         [Fact]
379         public static void RecursiveStringFormattingTest()
380         {
```

```
            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.

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

                var ab = links.GetOrCreate(a, b);
                var cb = links.GetOrCreate(c, b);
                var ac = links.GetOrCreate(a, c);

                a = links.Update(a, c, b);
                b = links.Update(b, a, c);
                c = links.Update(c, a, b);

                Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
                Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
                Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));

                Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                ↪  "(5:(4:5 (6:5 4)) 6)");
                Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                ↪  "(6:(5:(4:5 6) 6) 4)");
                Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
                ↪  "(4:(5:4 (6:5 4)) 6)");

                // TODO: Think how to build balanced syntax tree while formatting structure (eg.
                ↪  "(4:(5:4 6) (6:5 4)") instead of "(4:(5:4 (6:5 4)) 6)"

                Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
                ↪  "{{5}{5}{4}{6}}");
                Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                ↪  "{{5}{6}{6}{4}}");
                Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                ↪  "{{4}{5}{4}{6}}");
            }
        }

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

        #endregion

        #region Performance

        /*
        public static void RunAllPerformanceTests()
        {
            try
            {
                links.TestLinksInSteps();
            }
            catch (Exception ex)
            {
                ex.WriteToConsole();
            }

            return;

            try
            {
                //ThreadPool.SetMaxThreads(2, 2);

                // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    ↪  результат
                // Также это дополнительно помогает в отладке
                // Увеличивает вероятность попадания информации в кэши
                for (var i = 0; i < 10; i++)
                {
                    //0 - 10 ГБ
                    //Каждые 100 МБ срез цифр

                    //links.TestGetSourceFunction();
                    //links.TestGetSourceFunctionInParallel();
                    //links.TestGetTargetFunction();
```

```
                    //links.TestGetTargetFunctionInParallel();
                    links.Create64BillionLinks();

                    links.TestRandomSearchFixed();
                    //links.Create64BillionLinksInParallel();
                    links.TestEachFunction();
                    //links.TestForeach();
                    //links.TestParallelForeach();
                }

                links.TestDeletionOfAllLinks();

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

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

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

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

            GetBaseRandomLoopOverhead(linksStep);
            GetBaseRandomLoopOverhead(linksStep);

            var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);

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

            var loops = totalLinksToCreate / linksStep;

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

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

            ConsoleHelpers.Debug();

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

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

            ConsoleHelpers.Debug();

            ConsoleHelpers.Debug("C S D");

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

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

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

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

```
527         }
528
529         private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
    ↪  amountToCreate)
530         {
531             for (long i = 0; i < amountToCreate; i++)
532                 links.Create(0, 0);
533         }
534
535          private static TimeSpan GetBaseRandomLoopOverhead(long loops)
536          {
537              return Measure(() =>
538              {
539                  ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
540                  ulong result = 0;
541                  for (long i = 0; i < loops; i++)
542                  {
543                      var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544                      var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
545
546                      result += maxValue + source + target;
547                  }
548                  Global.Trash = result;
549              });
550          }
551           */
552
553          [Fact(Skip = "performance test")]
554          public static void GetSourceTest()
555          {
556              using (var scope = new TempLinksTestScope())
557              {
558                  var links = scope.Links;
559                  ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
                     ↪  Iterations);
560
561                  ulong counter = 0;
562
563                  //var firstLink = links.First();
564                  // Создаём одну связь, из которой будет производить считывание
565                  var firstLink = links.Create();
566
567                  var sw = Stopwatch.StartNew();
568
569                  // Тестируем саму функцию
570                  for (ulong i = 0; i < Iterations; i++)
571                  {
572                      counter += links.GetSource(firstLink);
573                  }
574
575                  var elapsedTime = sw.Elapsed;
576
577                  var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
578
579                  // Удаляем связь, из которой производилось считывание
580                  links.Delete(firstLink);
581
582                  ConsoleHelpers.Debug(
583                      "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                         ↪  second), counter result: {3}",
584                      Iterations, elapsedTime, (long)iterationsPerSecond, counter);
585              }
586          }
587
588          [Fact(Skip = "performance test")]
589          public static void GetSourceInParallel()
590          {
591              using (var scope = new TempLinksTestScope())
592              {
593                  var links = scope.Links;
594                  ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
                     ↪  parallel.", Iterations);
595
596                  long counter = 0;
597
598                  //var firstLink = links.First();
599                  var firstLink = links.Create();
600
601                  var sw = Stopwatch.StartNew();
```

```
602
603                 // Тестируем саму функцию
604                 Parallel.For(0, Iterations, x =>
605                 {
606                     Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
607                     //Interlocked.Increment(ref counter);
608                 });
609
610                 var elapsedTime = sw.Elapsed;
611
612                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
613
614                 links.Delete(firstLink);
615
616                 ConsoleHelpers.Debug(
617                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
618                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
619             }
620         }
621
622         [Fact(Skip = "performance test")]
623         public static void TestGetTarget()
624         {
625             using (var scope = new TempLinksTestScope())
626             {
627                 var links = scope.Links;
628                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
                    ↪  Iterations);
629
630                 ulong counter = 0;
631
632                 //var firstLink = links.First();
633                 var firstLink = links.Create();
634
635                 var sw = Stopwatch.StartNew();
636
637                 for (ulong i = 0; i < Iterations; i++)
638                 {
639                     counter += links.GetTarget(firstLink);
640                 }
641
642                 var elapsedTime = sw.Elapsed;
643
644                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
645
646                 links.Delete(firstLink);
647
648                 ConsoleHelpers.Debug(
649                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
650                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
651             }
652         }
653
654         [Fact(Skip = "performance test")]
655         public static void TestGetTargetInParallel()
656         {
657             using (var scope = new TempLinksTestScope())
658             {
659                 var links = scope.Links;
660                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
                    ↪  parallel.", Iterations);
661
662                 long counter = 0;
663
664                 //var firstLink = links.First();
665                 var firstLink = links.Create();
666
667                 var sw = Stopwatch.StartNew();
668
669                 Parallel.For(0, Iterations, x =>
670                 {
671                     Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
672                     //Interlocked.Increment(ref counter);
673                 });
674
675                 var elapsedTime = sw.Elapsed;
676
677                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
```

```
678
679                 links.Delete(firstLink);
680
681                 ConsoleHelpers.Debug(
682                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
683                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
684             }
685         }
686
687         // TODO: Заполнить базу данных перед тестом
688         /*
689         [Fact]
690         public void TestRandomSearchFixed()
691         {
692             var tempFilename = Path.GetTempFileName();
693
694             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
695             {
696                 long iterations = 64 * 1024 * 1024 /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
697
698                 ulong counter = 0;
699                 var maxLink = links.Total;
700
701                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
702
703                 var sw = Stopwatch.StartNew();
704
705                 for (var i = iterations; i > 0; i--)
706                 {
707                     var source =
    ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
708                     var target =
    ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
709
710                     counter += links.Search(source, target);
711                 }
712
713                 var elapsedTime = sw.Elapsed;
714
715                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
716
717                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
    ↪  Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
    ↪  counter);
718             }
719
720             File.Delete(tempFilename);
721         }*/
722
723         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
724         public static void TestRandomSearchAll()
725         {
726             using (var scope = new TempLinksTestScope())
727             {
728                 var links = scope.Links;
729                 ulong counter = 0;
730
731                 var maxLink = links.Count();
732
733                 var iterations = links.Count();
734
735                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
                    ↪  links.Count());
736
737                 var sw = Stopwatch.StartNew();
738
739                 for (var i = iterations; i > 0; i--)
740                 {
741                     var linksAddressRange = new
                        ↪  Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
742
743                     var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
744                     var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
745
746                     counter += links.SearchOrDefault(source, target);
747                 }
```

```
748
749                 var elapsedTime = sw.Elapsed;
750
751                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
752
753                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
                    ↪  Iterations per second), c: {3}",
754                      iterations, elapsedTime, (long)iterationsPerSecond, counter);
755             }
756         }
757
758         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
759         public static void TestEach()
760         {
761             using (var scope = new TempLinksTestScope())
762             {
763                 var links = scope.Links;
764
765                 var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
766
767                 ConsoleHelpers.Debug("Testing Each function.");
768
769                 var sw = Stopwatch.StartNew();
770
771                 links.Each(counter.IncrementAndReturnTrue);
772
773                 var elapsedTime = sw.Elapsed;
774
775                 var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
776
777                 ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
                    ↪  links per second)",
778                     counter, elapsedTime, (long)linksPerSecond);
779             }
780         }
781
782         /*
783         [Fact]
784         public static void TestForeach()
785         {
786             var tempFilename = Path.GetTempFileName();
787
788             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
789             {
790                 ulong counter = 0;
791
792                 ConsoleHelpers.Debug("Testing foreach through links.");
793
794                 var sw = Stopwatch.StartNew();
795
796                 //foreach (var link in links)
797                 //{
798                 //    counter++;
799                 //}
800
801                 var elapsedTime = sw.Elapsed;
802
803                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
804
805                 ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
    ↪  links per second)", counter, elapsedTime, (long)linksPerSecond);
806             }
807
808             File.Delete(tempFilename);
809         }
810         */
811
812         /*
813         [Fact]
814         public static void TestParallelForeach()
815         {
816             var tempFilename = Path.GetTempFileName();
817
818             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
819             {
820
821                 long counter = 0;
822
```

```
823                 ConsoleHelpers.Debug("Testing parallel foreach through links.");
824
825                 var sw = Stopwatch.StartNew();
826
827                 //Parallel.ForEach((IEnumerable<ulong>)links, x =>
828                 //{
829                 //    Interlocked.Increment(ref counter);
830                 //});
831
832                 var elapsedTime = sw.Elapsed;
833
834                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
835
836                 ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
    ↪  {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
837             }
838
839             File.Delete(tempFilename);
840         }
841         */
842
843         [Fact(Skip = "performance test")]
844         public static void Create64BillionLinks()
845         {
846             using (var scope = new TempLinksTestScope())
847             {
848                 var links = scope.Links;
849                 var linksBeforeTest = links.Count();
850
851                 long linksToCreate = 64 * 1024 * 1024 /
                    ↪  UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
852
853                 ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
854
855                 var elapsedTime = Performance.Measure(() =>
856                 {
857                     for (long i = 0; i < linksToCreate; i++)
858                     {
859                         links.Create();
860                     }
861                 });
862
863                 var linksCreated = links.Count() - linksBeforeTest;
864                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
865
866                 ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
867
868                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    ↪  linksCreated, elapsedTime,
869                     (long)linksPerSecond);
870             }
871         }
872
873         [Fact(Skip = "performance test")]
874         public static void Create64BillionLinksInParallel()
875         {
876             using (var scope = new TempLinksTestScope())
877             {
878                 var links = scope.Links;
879                 var linksBeforeTest = links.Count();
880
881                 var sw = Stopwatch.StartNew();
882
883                 long linksToCreate = 64 * 1024 * 1024 /
                    ↪  UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
884
885                 ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
886
887                 Parallel.For(0, linksToCreate, x => links.Create());
888
889                 var elapsedTime = sw.Elapsed;
890
891                 var linksCreated = links.Count() - linksBeforeTest;
892                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
893
894                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    ↪  linksCreated, elapsedTime,
895                     (long)linksPerSecond);
896             }
897         }
```

```

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

                ConsoleHelpers.Debug("Deleting all links");

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

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

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

        #endregion
    }
}
```

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

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

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

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

```
using Xunit;
using Platform.Converters;
using Platform.Memory;
using Platform.Reflection;
using Platform.Scopes;
```

```
using Platform.Data.Numbers.Raw;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Numbers.Unary;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Doublets.ResizableDirectMemory.Generic;

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

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

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

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

                var itself = links.Constants.Itself;

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

```
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